Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

A Pattern Language for Parallelizing Irregular Algorithms

Pedro Miguel Ferreira Costa Monteiro

Dissertação apresentada na Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa para Obtenção do grau de Mestre em Engenharia Informática

Orientador: Prof. Doutor Miguel Pessoa Monteiro

Lisboa

2010

Acknowledgements

The challenges proposed by a MSc dissertation are many and require not only hard work, but also constant support, care and attention. To those who have accompanied me, I leave my thanks:

None of this would have been possible without the unconditional support and encouragement of my family and friends. My mother and sister, who never lost faith in me in all this years. My nephews, whose thoughts gave me strength. My better half, who gave me a purpose. All my close friends, who have accompanied me in this long road, especially João and Tiago, the former for motivating me in the beginning of this journey and the latter for being with me until the end. Many thanks also to Nuno, my closest friend and without whose constant concern and motivation most of this would not have been accomplished. I must also thank all colleagues of the Centre for Documentation and Library, many of which I am proud to call my friends: Filomena, Salima, Lina, Rosário, Ana. All who have encouraged me and helped me reach this stage.

Special thanks are also merited by Professor Miguel Monteiro, my MSc supervisor, for his inspiration, support, and constructive criticism which always pushed me to do better.

For accompanying me through the development of this pattern language, I owe my thanks to Keshav Pingali and Mario Méndez-Lojo, of the Institute for Computational Engineering and Sciences of the University of Texas at Austin, and to João L. Sobral of Minho University, Portugal. They, together with Professor Pedro Medeiros and my supervisor made possible the internship at Texas, with which I have grown and matured, not only as an academic researcher, but as an individual. Special thanks go also to the rest of the Interns that accompanied me to Austin, to them go the merit of enduring me through that long month.

(Page intentionally left blank)

Resumo

Em algoritmos irregulares, as dependências e distribuições dos conjuntos de dados não podem ser previstas de forma estática.

Esta classe de algoritmos tende a organizar as computações consoante a localização dos dados em vez de paralelizar o controlo em múltiplas threads. Assim, as oportunidades para explorar paralelismo variam dinamicamente conforme o algoritmo altera a dependência entre os dados. O que leva a que a paralelização eficaz desses algoritmos exija novas abordagens que tenham em conta essa natureza dinâmica.

Esta dissertação procura resolver o problema da criação de implementações paralelas eficientes através de uma abordagem que propõe a extracção, análise e documentação de padrões de concorrência e paralelismo presentes na framework Galois para paralelismo de algoritmos irregulares. Padrões são representações formais de uma possível solução de um problema que surge num contexto bem definido de um domínio específico.

Os padrões referidos são documentados através de uma linguagem de padrões que evidência um conjunto de padrões inter-dependentes, que compõe um modelo de uma solução que pode ser reutilizada sempre que um problema especifico surja.

Palavras-Chave:

- Linguagem de Padrões
- Algoritmos Irregulares
- Computação Paralela
- Engenharia Reversa
- Molduras Orientadas a Objectos

(Page intentionally left blank)

Abstract

In irregular algorithms, data set's dependences and distributions cannot be statically predicted. This class of algorithms tends to organize computations in terms of data locality instead of parallelizing control in multiple threads. Thus, opportunities for exploiting parallelism vary dynamically, according to how the algorithm changes data dependences. As such, effective parallelization of such algorithms requires new approaches that account for that dynamic nature.

This dissertation addresses the problem of building efficient parallel implementations of irregular algorithms by proposing to extract, analyze and document patterns of concurrency and parallelism present in the Galois parallelization framework for irregular algorithms. Patterns capture formal representations of a tangible solution to a problem that arises in a well defined context within a specific domain.

We document the said patterns in a pattern language, i.e., a set of inter-dependent patterns that compose well-documented template solutions that can be reused whenever a certain problem arises in a well-known context.

Keywords:

- Pattern language
- Irregular Algorithms
- Parallel Computing
- Reverse Engineering
- Object-Oriented Frameworks

(Page intentionally left blank)

Table of Contents

List of Figures

(Page intentionally left blank)

Code Listings

(Page intentionally left blank)

1 Introduction

This dissertation presents and documents a pattern language for parallelizing irregular algorithms. The body of work produced in this dissertation builds upon the work produced at the University of Texas in Austin, namely the Galois framework, and was partly supported by the project Parallel Refinements for Irregular Applications (UTAustin/CA/0056/2008) funded by FCT-MCTES and European funds (FEDER).

1.1 Motivation

Gustafson's law [1] states that any sufficiently large problem can be efficiently parallelized and has proven that parallelization is an effective way to accelerate the processing of massive data. However, in practice not all applications are easily parallelized and finding the right programming model and architecture for a given algorithm is quite challenging in the multicore era. Issues such as race conditions, communication, scalability, load balancing, data distribution, and locality further add to the effort of achieving efficient parallel programs.

Many approaches, methodologies, libraries, languages and frameworks have been devised and these are, for the majority of algorithms, able to produce efficient parallel implementations. Aside from those "regular" algorithms, not much attention has been granted to the so-called irregular algorithms and applications [2]. The parallelization of *irregular algorithms* [3-4] is constrained by irregular accesses to dynamic pointer-based data structures whose data-dependence set can only be uncovered at run-time. In this context irregular algorithms pose a challenging problem to current parallelization methods and techniques.

By developing the pattern language presented in this dissertation, we aim to increase the knowledge base of best practices in parallel programming of irregular algorithms and reduce the effort of producing new core synchronization concepts and other parallelism related components. To that end, we propose to extract, analyze and document concurrency patterns from Galois, a parallelization framework for irregular algorithms.

Patterns capture formal solutions to specific problems, while maintaining a level of abstraction similar to that of design models (e.g., UML) and above source code. This way, patterns support a high-level form of reuse, which is independent from language, paradigm and hardware. Identifying and documenting patterns of complex concurrent software problems is one of key practices that will allow concurrent software development to be established as an engineering discipline – one which requires thorough systematic understanding and documentation of successful practices [5].

Pattern catalogs and languages for software design represent a widely prolific area of development, partly due to the renowned Gang of Four catalog of object-oriented design patterns [6]. From this first approach, patterns became popular in the field of reusable design, branching different application areas such as object-oriented programming [7], aspect-oriented programming [8] framework design [9-10], software architecture [11-12], components [13], machine learning [14-15] and even patterns about patterns [16-17].

1.2 **Contributions**

To the best of our knowledge, this pattern language is the first to address specific solutions to the problems of irregular algorithms. We have described a set of ten patterns for parallelizing irregular algorithms. These present knowledge derived from the Galois framework, which was in turn inherited from years of insights and experiences on parallel software development. Additionally they present a high-level approach that allows for the dissemination of knowledge that before was property of expert parallel software developers. Furthermore, the set of patterns is documented as a *Pattern Language*, i.e. set of inter-dependent patterns. Pattern languages guide *pattern-oriented software development*, such that choosing to use one pattern will eventually direct the software developer to use another related pattern. Following the sequence of pattern dependences will eventually lead to an efficient parallelization of an irregular algorithm.

1.3 **Structure**

The remainder of this dissertation is organized as follows:

- *Chapter 2* overviews the problem being tackled by providing an overview of the concept of irregular algorithms and of amorphous data-parallelism, the specific form of parallelism that can be extracted from this class of algorithms (section 2.1).

Finally, some insight is given as to how these algorithms can be categorized in order to provide reusable abstractions (section 2.2).

- *Chapter 3* presents the Galois framework and provides a general overview of the Galois Execution Model (section 0). It follows by describing worklist-based implementation of irregular algorithms in Galois (section 3.2). This chapter concludes by presenting some Galois specific terminology in section 3.3 .
- *Chapter 4* describes some irregular algorithms and provides the appropriate implementations in the Galois framework. The set of irregular algorithms is comprised of: *Delaunay Triangulation*, an algorithm for the generation of triangular meshes (aection 4.1), *Preflow-Push,* a max-flow algorithm (section 4.2), *Sparse Cholesky Factorization*, a traditional linear algebra algorithm for matrix factorization (section 4.3) and *Kruskal's Minimum Spanning Tree* (section 4.4).
- *Chapter 5* describes the Archlight Eclipse plugin, which was implemented to extract metrics from Galois and determine the viability of our pattern mining approach.
- *Chapter 6* describes the concept of pattern (section 6.1), pattern languages (section 6.2) and introduces the general form as style of patterns description (section 6.2).
- *Chapter 7* presents the pattern language and contextualizes it by proposing some abstract terminology used in the pattern descriptions (section 7.1). The next three sections document the set of patterns that compose our pattern language: *Structure Patterns* in section 7.2, *Execution Patterns* in section 7.3 and *Optimization Patterns* in section 7.4. Each section is further refined into the specific patterns.
- *Chapter 8* presents an overview of related work in the field of pattern languages for parallel computing.
- *Chapter 9* describes a summary of contributions and a discussion of future work.
- *Chapter 10* presents the bibliographical references used in this dissertation.

(Page intentionally left blank)

2 Irregular Algorithms

The programming community is not always in harmony and although algorithm irregularity is frequently considered in the literature, there is no consensual definition of what in fact constitutes an irregular algorithm. Some authors refer to irregular algorithms as those on which data is structured as multidimensional arrays and referenced through array indirections and indexed values [18-19]. Other authors consider that the irregularity factor is due to the dynamism of pointer-based data-structures [20-22]. There are others even that consider algorithms to be irregular due to input dependent communication patterns [23] or irregular data distribution among the processors [24].

Our view is that, although there is no consensus on a single definition, in fact there is a clear pattern among different descriptions. Most references to irregularity as a problem of indirect access to data can be found in articles published until around the mid 1990s, roughly when object-oriented programming became widespread in the programming community [25]. From hereafter, object-orientation, and essentially pointer-based programming, became the tool of choice for the implementation of most algorithms, including irregular, giving rise to the second definition. The following two definitions arise from the fact that, in pointer-based data-structures, growth can be unpredictable, which will easily lead to irregular distributions of data among partitions and of tasks required to handle such data.

We claim that the problem of irregularity can be defined in a more abstract way as a problem of *unpredictability of data dependences*. Having stated that, is no amount of static planning can account for the unpredictability of run-time behavior when considering *irregular computational dependences*; no statically-defined fine-grain locking mechanism can protect an unpredictable dynamic set of data from concurrent access; no non-dynamic balancing algorithm can account for irregular data distribution. These irregular problems arise especially in the scientific domain as most of simulation algorithms present data unpredictability and irregularity. Examples of such algorithms include sparse matrix computations, computational fluid dynamics, image processing, molecular dynamics simulations, galaxy simulations, climate modeling and optimization problems [26].

2.1 **Amorphous data parallelism**

Irregular algorithms are *data-parallel algorithms* [27] and essentially perform multiple operations on large data-sets, organizing computations in terms of data locality instead of parallelizing control in multiple threads. When data-set's dependencies and distributions are unpredictable and dynamic, as in the case of implementations of irregular algorithms, the amount of parallelism that can be achieved varies according to how the algorithm changes its data dependences. As such, effective parallelization of irregular algorithms requires new approaches that account for the dynamic nature this class of algorithms.

Amorphous data-parallelism [20], is the type of parallelism that arises when the data-sets being iterated have no fixed shape or size, i.e. are *amorphous*. This means that the amount of available opportunities for concurrency-free parallelism changes throughout the execution of the algorithm. It is an example of data-parallelism in which simultaneous operations may interfere with each other and in which the underlying data-structure might be modified.

2.2 **Categorization of irregular algorithms**

Pingali *et al* [20] present a general categorical division of irregular algorithms that allows the reuse of patterns of parallelism and locality common to these algorithms. This categorization, shown in **Fig. 1**, provides a simple yet expressive way to address the implementation of irregular algorithms.

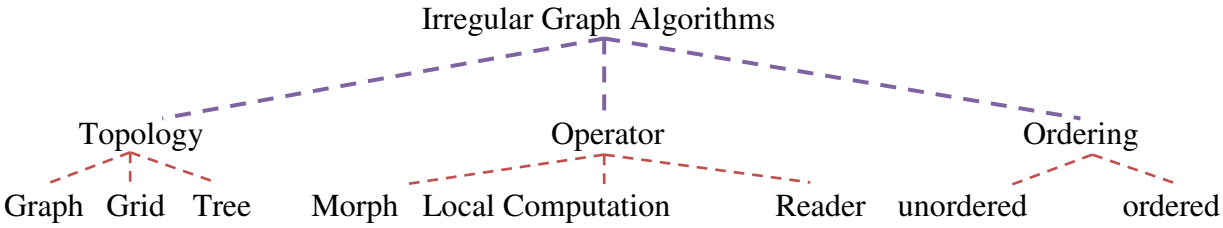


Fig. 1 – Categorical division of Irregular Graph Algorithms.

A detail description of each category is described next:

The *topology category* pertains to the overall shape of the data-structure. The general form is that of a simple graph (**Fig. 2-a**). The two other types of graphs are special forms of sparse graphs, which have relatively few edges. Trees are special graphs where there are no cycles

and the starting node is called root (**Fig. 2-b**). A grid is a graph in which every node is connected to four neighbors (**Fig. 2-c**).

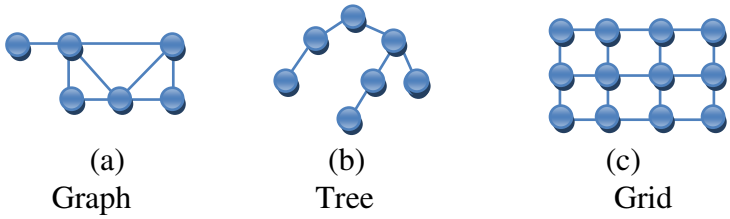


Fig. 2 – Different graph topologies in irregular algorithms

The *computational operator* is classified in terms of how the active node neighborhood is changed by the action of the operator (**Fig. 3**). The operator of an algorithm can be one of three types: *Morph, Local Computation* and *Reader*.

- **Morph algorithms**

  Morph algorithms considerably change the structure of the graph by adding or removing nodes and edges. This can be done by either *coarsening, refinement* or *reduction*. *Coarsening algorithms* iteratively collapse adjacent nodes together until the graph forms a coarser sub-graph. *Boruvka's MST algorithm*, for example, builds the minimum spanning tree bottom-up by coarsening [28]. Contrary to coarsening, *refinement algorithms* iteratively generate the output graph from a subset of the nodes. This is the case of *Delaunay Triangulation* [4, 29] and *Delaunay Mesh Refinement* [30]. *Reduction algorithms* are similar to coarsening but simply remove nodes and edges from the graph, not actually contracting elements [31-32].

- **Local Computation Algorithms**

  This class of algorithm operator does not modify the underlying graph structure but instead updates its labels and data elements (e.g. *Preflow-push algorithm* [33]).

- **Reader Algorithms**

  This type of operator only reads the graph and does not modify it in any way.

  The type of operator is inferred to be the type with most computational impact, i.e., if an irregular algorithm performs a read and a morph, the type of operator present in the algorithm is considered to be a morph operator.

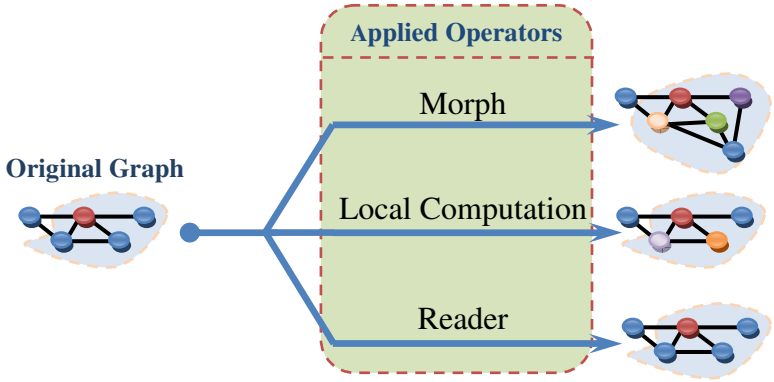


Fig. 3 - Action of the different Computational Operators

The *ordering of execution* of an iteration must be chosen so as to avoid data-races and consistency problems. *Unordered execution* is the case where the sequence in which nodes are executed has a non-deterministic aspect, meaning that output is independent of this same sequence. *Ordered execution* implies that the output of the algorithm is influenced by the sequence by which nodes are executed. The order might be full or only partial but nevertheless parallelization of execution on ordered sets is difficult to implement, because it requires a more restricting scheduling policy. A sense of ordering can be represented by directional graphs.

3 The Galois Framework

This chapter presents the Galois framework [2, 34] which tackles the problems that arise from trying to parallelize irregular algorithms and applications (chapter 2). It does so by building upon the categorization presented in section 2.1 .

As stated in chapter 2, irregular algorithms are often associated with the scientific community. The effort needed to create efficient parallel versions of these algorithms is not easily managed by non-expert scientific programmers, which are more accustomed to view problems in a sequential manner [35]. The Galois framework's main objective is to solve this problem by using an *optimistic* approach to parallelization that doesn't require the programmer to perform any major changes to the base sequential implementation of the algorithm. Furthermore, the Galois framework provides only a small number of syntactic parallelization constructs, leaving the bulk effort of parallelization to the underlying runtime system.

The *Optimistic Iteration* approach consists of running parallel tasks while assuming that there is no concurrency in data access and that data dependences are maintained throughout the execution. If no race condition occurs, all operations follow regular parallel execution, committing its updates and synchronizing at the end. However, the system makes dependence checks and if a violation occurs, the task that detected the violation is halted and rolled back to its initial state. Upon rollback, every update that task performed on shared objects is undone and the task starts anew. Using *transactional semantics* helps reduce some of the overheads of lock-based shared-memory synchronization [36]. **Fig. 4** shows a small example of Galois' execution model.

3.1 Galois execution model

The Galois is comprised of three interconnected components: *user code, library classes* and *runtime system*. User code is parallelized by the library classes which are in turn managed by the runtime.

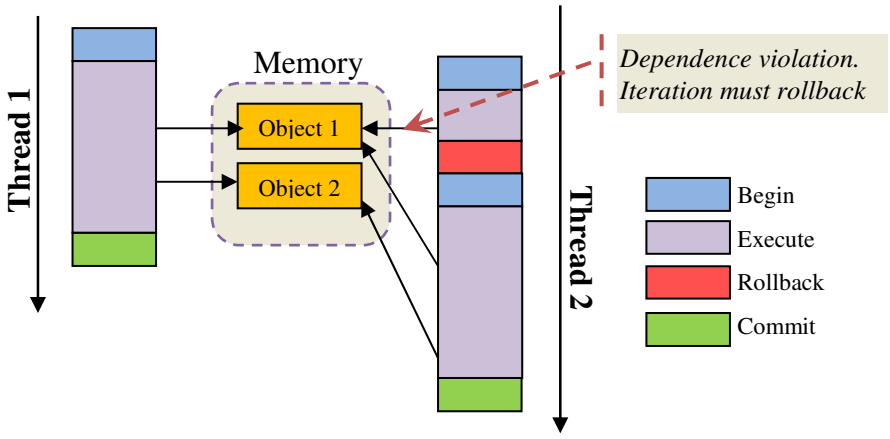


Fig. 4 – Galois optimistic execution model.

Programming model

User code is the code a programmer would use to create and manipulate shared objects, expressing a given irregular algorithm. It is based on a programming model that uses set iterators to introduce optimistic parallelism. This approach helps programmers abstract the algorithm from the parallelization concerns and allows parallel algorithms to have sequential-like semantics. The semantic of the set iterators (**Fig. 5**) is independent of the type of data set being iterated over. This means that the programmer must only concern himself with the algorithm ordering constraints and not with the underlying programming model.

```
foreach( element in set ){
    doSomething(element)
}
```

Fig. 5 – Foreach set iterator in Galois.

These data sets iterators act as data-oriented worklists in which the order of iteration committal is constrained by the order of the elements in the data set. In case of *unordered sets*, no particular ordering is enforced. Even if there are dependences among iterations, the result is the same for whichever order the iterations occur. *Ordered sets* restrict the order of committal to that of the partial ordered set they are iterating. Both sets are unbounded,

meaning that at any moment during the execution of an iteration a new element can be added to the set.

Galois is based on an object-oriented shared memory model with cache coherence. Direct memory accesses are not allowed and data is accessed by invoking object methods, which is easier for programmers.

Class library

The *Galois class library* provides method and shared-object implementations to support the implementation of irregular algorithms in Galois. Furthermore, these classes specify how parallel manipulation of object-oriented data can be achieved and provide locality and correctness abstractions for data-structures.

Galois runtime

The runtime of the Galois framework is responsible for issuing iterations to threads and ensuring their subsequent committal or, in case of conflicting iterations, enforcing rollback operations.

3.2 Worklist-based algorithms

Worklists are special data-structures which hold thread-executable units of work, often referred to as tasks. This structure is meant to be accessed in a synchronized way by threads, which retrieve independent tasks and process them concurrently with other task-executing threads. However, in Galois, the runtime system has a *scheduler* which is responsible for fetching work from the set iterators and creating optimistic parallel iterations. In this instance, set iterators act as data-driven worklists.

Irregular algorithms have two characteristics that make them ideal for implementations using worklist parallelism:

- Execution model is centered on iterative processing of data in a loop.
- Each iteration might add more elements to the iteration space.

Thus, tasks can be abstracted from loops by identifying independence in the set of iterations. The amount of tasks available depends on the granularity of the minimum set of independent

iterations. When an iteration produces more work, a new task is added to the worklist. A pseudo-code example of a basic worklist algorithm can be seen in **Listing 1.**

```
1   Worklist wl = //create worklist and initialize it
2   While(wl notEmpty()){
3       Element el = wl.getNext();
4       //perform computations using element
5       work = compute(el)
6       if(work!=null)
7           wl.add(work);
8   }
```

Listing 1 – Iterative worklist algorithm.

3.3 **Galois terminology**

For a full grasp of the Galois framework, the programmer must understand the abstractions used to separate the actual implementation from the algorithm-specific terminology. In this context, and considering that Galois' data-structure abstraction is that of a graph, we refer to an *active node* as the node where computation occurs. The *neighborhood* of an active node is composed of the set of nodes that are accessed or modified by the active node's computation. **Fig. 6** shows how these concepts are represented in a graph topology.

Fig. 6 – Galois abstractions for Irregular algorithms

The concept of *amorphous data-parallelism* (section 2.1 stems from this definition as the type of parallelism that can be achieved by parallel processing active nodes subject to neighborhood and ordering constraints. This concept is abstract but it allows us to directly reference parallelism in irregular applications as a special application of general data-parallelism.

4 Irregular Algorithms in Galois

In this chapter we shall discuss a few algorithms implemented in Galois so as to provide some insight into some of the concepts previously described. To implement any sort of irregular algorithm in Galois, the programmer must always introduce the following changes to the code:

Use Galois Classes

The Galois Framework provides the programmer with a set of data-structures with which to express the algorithm. These are essentially so that Galois' runtime is able to recognize how to handle data objects and process the algorithm. These data-structures are implemented around a Graph interface, providing support for directed and undirected graphs, as well as complex, simple and indexed edges. Other shared data-structures and object classes such as a Map, Collection, Set and Accumulator class, provide synchronized runtime logic and are able to be subclassed to suit the user's needs.

Use Galois Worklists

The Galois framework is directed at worklist implementations of irregular algorithms (as discussed in section 3.2), since this is the ideal way in which to explore available amorphous data-parallelism in this type of algorithms. A parallel implementation of an algorithm using a worklist is usually more balanced than other implementations because each thread fetches work as needed. This means that the worst case happens when there is no more work left to be processed but one thread is still processing its task.

Iterations are guided by worklists, which impose ordering constraints, if they exist. Thus, the user must select and instantiate the appropriate worklist for the algorithm. Galois provides three types of worklists – ordered, partitioned unordered and unpartitioned unordered – that are instantiated via the *GaloisWorklistFactory* class.

Use Galois Foreach loops

Galois Iterations require the user to identify the main loops in the algorithm, the ones that guide parallelism, and convert them into *foreach* loops. As described in programming model in section 0, foreach loops iterate over the elements of the worklist.

Having processed the set of transformations described above, the programmer would eventually reach a base Galois implementation of a worklist algorithm similar to the one depicted in **Listing 2.**

```
1  Graph g = // initialize with input graph
2  Worklist wl<Elem> = // create worklist of desired type
3  //"Elem" is the type of element that composes a task
4  wl.add( elements ); // populate with initial tasks
5  foreach( Elem e in wl){
6       Elem work = //process e
7       if ( work != null) {
8          wl.add( work);
9       }
10 }
```

Listing 2 – Worklist algorithm in Galois.

4.1 Delaunay Triangulation Algorithm

Delaunay's Triangulation Algorithm [4, 29], also referred to as *Delaunay Mesh Generation,* is an algorithm for the generation of a mesh of triangles for a given set of points. In order to generate valid triangulations, every triangle in the generated mesh must fulfill the *Delaunay property*. This property states that given a circumference that intersects every triplet of points, no other point belonging to the mesh is located inside the circumference. This algorithm takes an input set of points in 2D space and as a first step surrounds all points with a single triangle. Then, iteratively picks a single point, determines its involving triangle and splits the triangle in three new triangles, with the selected point as focal node. It then follows by checking the Delaunay property and, if it detects a violation, flips the common edge to produce a valid triangulation. An example is given in **Fig. 7** and Galois based implementation code is described in **Listing 3**.

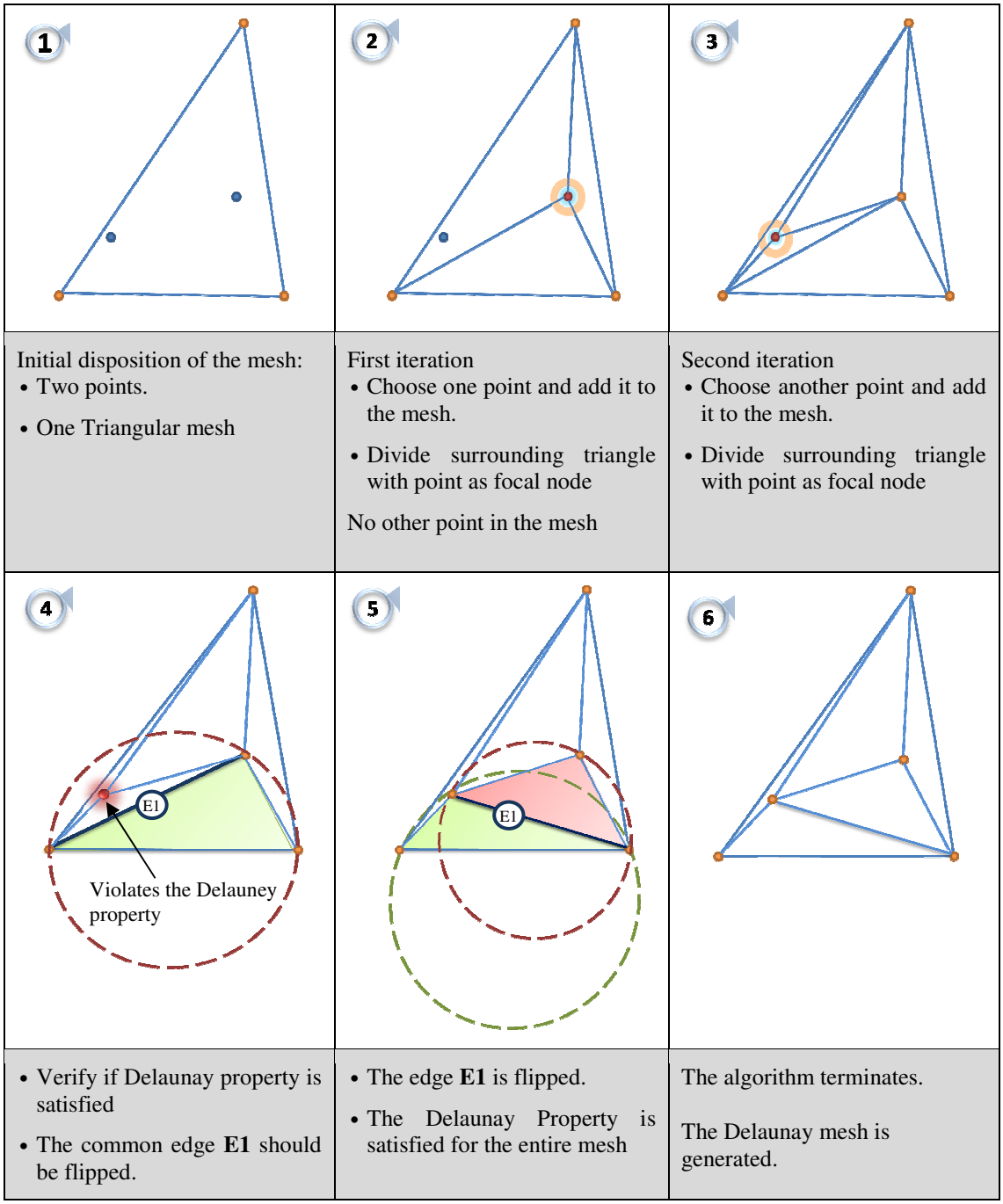


Fig. 7 – Example execution of the Delaunay Triangulation algorithm

Implementation in Galois

In Galois, the *Delaunay triangulation algorithm's* mesh is represented as a graph structure where each node represents a triangle and edges represent adjacencies between those same triangles. On selecting an active node, in this case one of the points to be added to the mesh, the neighborhood consists on the set of triangles affected by the eventual splitting and edge flipping activities. Using triangles as nodes reduces the amount of nodes locked in the

processing of an active node and reduces the size of the graph while maintaining a tighter coupling of the data dependences.

An example of the implementation of this algorithm in Galois is described in **Listing 3** and further classification of this algorithm according to the categorization of section 2.1 is summarized in **Fig. 8**.

```
1    Mesh m = // initialize with one surrounding triangle
2    Set points = // read points to insert
3    Worklist wl;
4    wl.add( points);
5    foreach( Point p in wl){
6       Triangle t = m.surrounding(p);
7       Triangle newSplit [3] = m.splitTriangle( t, p);
8       Worklist wl2;
9       wl2.add(edges( newSplit));
10      foreach( Edge e in wl2){
11          if ( !isDelaunay(e)) {
12              Triangle newFlipped [2] = m.flipEdge(e);
13              wl2.add( edges( newFlipped))
14          }
15      }
16   }
```

Listing 3 – Delauney Triangulation algorithm in Galois.

Topology	Graph (undirected)
Operator type	Morph
Ordering	Unordered

Fig. 8 – Classification of the Delaunay Triangulation algorithm.

4.2 **Preflow-push Algorithm**

Max-Flow problems [37] consist on finding the maximum flow from a source node to a sink node through a directed graph. Loosely put, this kind of problem can be interpreted as "what is the maximum amount of liquid that can be pumped through a network of pipes", where the pipes are the edges in the graph. Each edge of the graph has a fixed capacity that represents the maximum amount of flow able to pass through that edge. The general idea underlying this algorithm is that the source is continuously pushing a steady flow to all its downstream neighbors and the flow must find its way to the sink without invalidating the capacity constraints. Furthermore, it has to guarantee that the amount of flow entering a node is equal to the amount of flow leaving that same node – or what is known as *flow-conservation property*. When no more flow can be pushed in the direction of the sink, the excess flow of a node is pushed back towards the source and must find a new pathway to the sink. There are many algorithms to solve this particular problem, one of which is the *Preflow-push algorithm* [33].

This algorithm's name derives from the fact that the algorithm does not maintain the flow-conservation property and instead relies on the notion of a *preflow*, which states that the amount of flow entering a node can at times be more than the total flow leaving that same node. This preflow property defines which nodes have an excess flow and therefore need to be analyzed and processed by the algorithm. Nodes have a hierarchical structure based on a positive height value label, being that the source has height equal to the number of nodes and the sink has height zero. Every other node begins with height equal to one. Throughout the execution of the algorithm, nodes having lower height values are considered to be closer to the sink than its "taller" neighbors. This way, flow is always pushed downstream, in the direction of the sink. Once the algorithm is processing a node, it tries to push flow to a neighboring node that has not reached its total excess capacity and has a lower height value. If there are no valid downstream nodes, then the node is *relabeled*, that is, its height is incremented until there is at least one available node to which flow can be pushed.

When all edges have reached its maximum flow capacity on the paths that lead to the sink, a max-flow has been found. The algorithm then pushes all remaining excess flow towards the source, until the amount of excess flow is equal to zero for every node in the network, and terminates.

A short example of a complete execution of the algorithm is shown in **Fig. 9**.

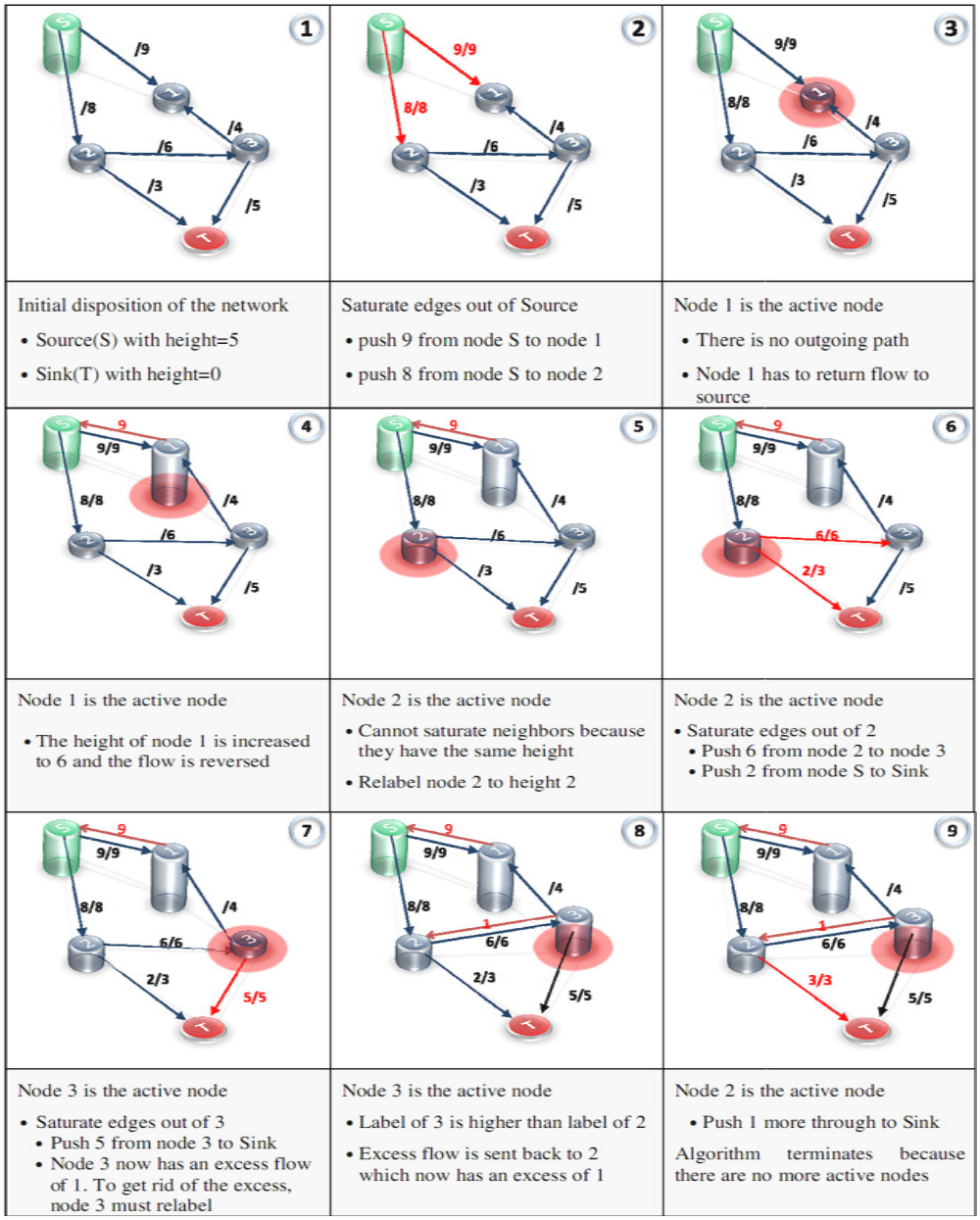


Fig. 9 – Example execution of the *Preflow-Push* algorithm

Implementation in Galois

In Preflow-Push, we define an active node as a node that has some excess flow. These are the nodes that will be added processes and therefore added to the worklist dynamically. Each iteration of the foreach then selects an active node and two activities are performed: Push and Relabel. These conform to a local computation operator type, in which they only modify the values stored by the nodes. For locking purposes, the neighborhood of an active node consists

of all its downstream neighboring nodes. Further classification of this algorithm according to the categorization of section 2.1 is summarized in **Fig. 10**.

Topology	Graph (directed)
Operator type	Local Computation
Ordering	Unordered

Fig. 10 – Classification of the Preflow-Push algorithm.

This behavior is introduced by the Preflowpush class, which provides the base algorithm implementation. An example of this implementation is described in pseudo-code in **Listing 4**.

```
1   Worklist wl = new Worklist( graph) //create worklist
2   foreach( Node node: wl){
3      //try to relabel the node
4      graph.relabel( node);
5      //try to push flow to every neighbor
6      for( Neighbor ng : graph.getNeighbors( node)){
7           if( graph.canPushFlow(node, ng)){
8                graph.pushFlow(node, ng);
9                if ( ! ng.isSourceOrSink())
10                    wl.add( ng);
11               if ( ! node.hasExcess())
12                    break;
13               }
14          }
15          if ( node.hasExcess())
16          wl.add( node);
17     }
```

Listing 4 – PreflowPush algorithm in Galois.

4.3 **Sparse Cholesky Factorization Algorithm**

Cholesky's factorization [38-39], also known as Cholesky decomposition is a linear algebra method that transforms a matrix into a factor of a unique lower triangular matrix. The general form of this factorization is $A = LL^T$, where:

- A is a symmetrical positive definite matrix. That is, all it's diagonal entries are positive and for every non-zero vector $x \in \mathbb{R}^n$, where x^T denotes the transpose matrix, $x^T A\, x > 0$.
- L is a lower triangular matrix, where by lower triangular matrix we mean a matrix with every entry above the main diagonal equal to zero.
- L^T is the transpose of the L matrix and therefore an upper triangular matrix with every entry below the main diagonal equal to zero.

As an algorithm, Cholesky has irregular data accesses and traditionally operates on a matrix data-structure, a property it inherits from linear algebra. There are several variations of Cholesky's factorization but one of the most commonly used, due to its simplicity and the use of sparse matrixes is the *Sparse Column-Cholesky factorization algorithm*. The column-oriented version of the *Cholesky factorization algorithm* is shown in **Listing 5**.

```
1 Matrix [rows] [columns] m;
2 for ( int col in columns){
3    for( int row in rows){
4       if(m [row] [col] != 0)
5         m [] [col] -= m [col] [col] * m [col] [row] * m
          [] [row];
6    }
7    //divide column m [] [col] by the diagonal
```

Listing 5 – Sparse Column-Cholesky factorization algorithm.

Implementation in Galois

Cholesky's column-oriented algorithm is pretty simple and straightforward but cannot be efficiently parallelized in a data-parallel manner. Therefore, given that the matrix is sparse and can be efficiently mapped onto a graph data-structure, adding the changes described in the beginning of this chapter, one would attain an algorithm identical to the one described in **Listing 6**.

```
1  //get sparse matrix
2  Graph g = //make graph from sparse matrix
3  foreach (Node node in g){
4      //divide column by the "diagonal"
5      for(Edge edge in g.getOutEdges(node)){
6          edge.data /= factor
7      }
8      //divide edges by a factor
9      for(Node node2 in neighbors(node)){
10         for(Node node3 in neighbors(node)){
11             edge = g.getEdge (node2, node3);
12             if((node2==node3) and notSeen(edge)){
13 //same as m [] [col]-= m [col] [col]* m [col] [row]* m
   [] [row];
14                 v2 = g.getEdge(node,node2).value;
15                 v3 = g.getEdge(node,node3).value;
16                 edge.data -= v2*v3;
17             }
18         }
19     }
20     //Add result to answer Graph
```

Listing 6 – Galois' Graph-based Cholesky factorization algorithm.

In this algorithm, the active nodes are the ones present in the sparse graph, corresponding to the non-zero elements in the original matrix. The neighborhood of the active node is the actual edge neighbors of that same node, except the ones already processes. This is identical to the original algorithm since iterating over the N nodes is equivalent to iterating over the columns of the matrix in the *Column-Cholesky* version (see **Listing 5**). Further classification of this algorithm according to the categorization of section 2.1 is summarized in **Fig. 11**.

Topology	Graph (undirected)
Operator type	Reader (in relation to the input graph) Morph (in relation to the output graph)
Ordering	Unordered

Fig. 11 – Classification of the Sparse Cholesky Factorization algorithm.

4.4 **Kruskal's Minimum Spanning Tree Algorithm**

Kruskal's algorithm [37] finds minimum spanning trees(MST), that is, given a graph it finds a tree that is composed of a set of edges such that:

- Every node in the graph is connected to at least one edge in the set.
- The total weight of the set of edges is less than or equal to the total weight of every other possible spanning tree.

Kruskal's MST is a special case of a more general problem called the *union-find problem* [40]. A union-find is a data-structure that represents a set of disjoint non-empty sets. There is a wide variety of implementations for the *Kruskal's MST* problem, but we shall only refer to the union-find implementation variant.

The general conceptualization of this algorithm first creates a union-find and populates it with the nodes in the graph, on disjoint non-empty set for each node. The edges of the graph are then placed in a priority queue, ordered by increasing edge weight. The algorithm then follows by iterating over an ordered queue containing every edge in the graph, in order of increased weight. In every iteration, the edge with the lowest weight is removed from the queue and if its connecting nodes belong to different sets, both sets are joined by creating an edge between those nodes in the union-find. If the nodes already belong to the same set, the edge is discarded and a new iteration commences. When all edges have been removed from the queue, the algorithm completes and the union-find now represents the minimum spanning tree of the original graph.

Galois implementation

In the Galois implementation of this algorithm, active elements are the edges of the graph, represented in an ordered worklist. For each iteration, the neighborhood of an active edge is composed of its connected nodes and all elements in the sets to whom the nodes. The union-find can be created by subclassing or wrapping one of the provided set of Galois graph classes. An example of the implementation of this algorithm in Galois is described in **Listing 7** and further classification of this algorithm according to the categorization of section 2.1 is summarized in **Fig. 12**.

Topology	Graph (undirected)
Operator type	Reader (in relation to the input graph) Morph (in relation to the union-find)
Ordering	Ordered

Fig. 12 – Classification of Kruskal's MST algorithm.

```
1 Graph g = // read in graph
2 MST mst = new MST( );
3 UnionFind uf = new UnionFind();
4
5 foreach( Node n in g ){
6    uf.create(n);//create new set
7 }
8 foreach( Edge e in g ){//ordered by weight
9    Node n1 = e.getHead();
10   Node n2 = e.getTail();
11   if( uf.find(n1)!=uf.find(n2)){
12      uf.union(n1,n2) ;
13      mst.add(e);//put e in MST
14   }
15}
```

Listing 7 – Galois implementation of Kruskal's MST

(Page intentionally left blank)

5 Arclight Plugin

Paramount to the task of identifying the concurrency patterns in the Galois framework, was the analysis of just how much Galois-specific code was present in the implemented algorithms. Our proposal was to identify the different concerns present in Galois implementations of algorithms and measure the amount of tangling present. A *concern*, according to Robbillard [41] is any type of special consideration about the software being implemented. In this case, we identified a set of five code concerns related to Galois: Algorithm code, Galois prologue code, Galois epilogue code, Galois interlogue code and Miscellaneous code.

- Algorithm code concerned the actual algorithm structure, whether implemented in Galois or in whichever other framework or language. The Galois-specific data-structures was also regarded as belonging to this concern since they only replace the previous data-structures.
- Galois prologue code concerned code that was needed to instantiate and initialize a Galois implementation.
- Galois epilogue code concerned post-algorithmic operations. The majority of this type of code concerned result verification code which was in fact irrelevant for the task at hand and so this code was also tagged as miscellaneous code.
- Galois interlogue code concerned Galois specific code that was interleaved with algorithmic code. This usually meant optimization related code.
- Miscellaneous code concerns non-essential code, such as comments, variable declaration, etc.

Six concern exploration tools were considered to the task of marking and exploring the concerns in the code of Galois algorithms: *FEAT* [42], *ConcernMapper* [43], *Sextant* [44], *JQuery* [45], *JTransform* [46] and *Inari [47]*. However, while all allowed code navigation and

concern identification, with varying degrees of efficiency, none presented the capability to extract metrics from the concerns.

To this task, *ArchLight*, an eclipse code tagging plugin, was implemented. This plugin consisted on a specialized toolbar (**Fig. 13**) that allowed the coloring of different concerns present in the code and the application of sizing metrics.

The plugin consisted in a toolbar that allowed us to colorize the code according to five types of concerns present in the set of Galois algorithms' code.

Fig. 13 – ArchLight plugin toolbar.

Fig. 14 shows how the correspondence between the coloring and the different concerns was accomplished.

A	Algorithm Code
P	Galois prologue code
M	Miscellaneous code
E	Galois epilogue code
I	Galois Interlogue code

Fig. 14 – ArchLight Eclipse Plugin concern coloring.

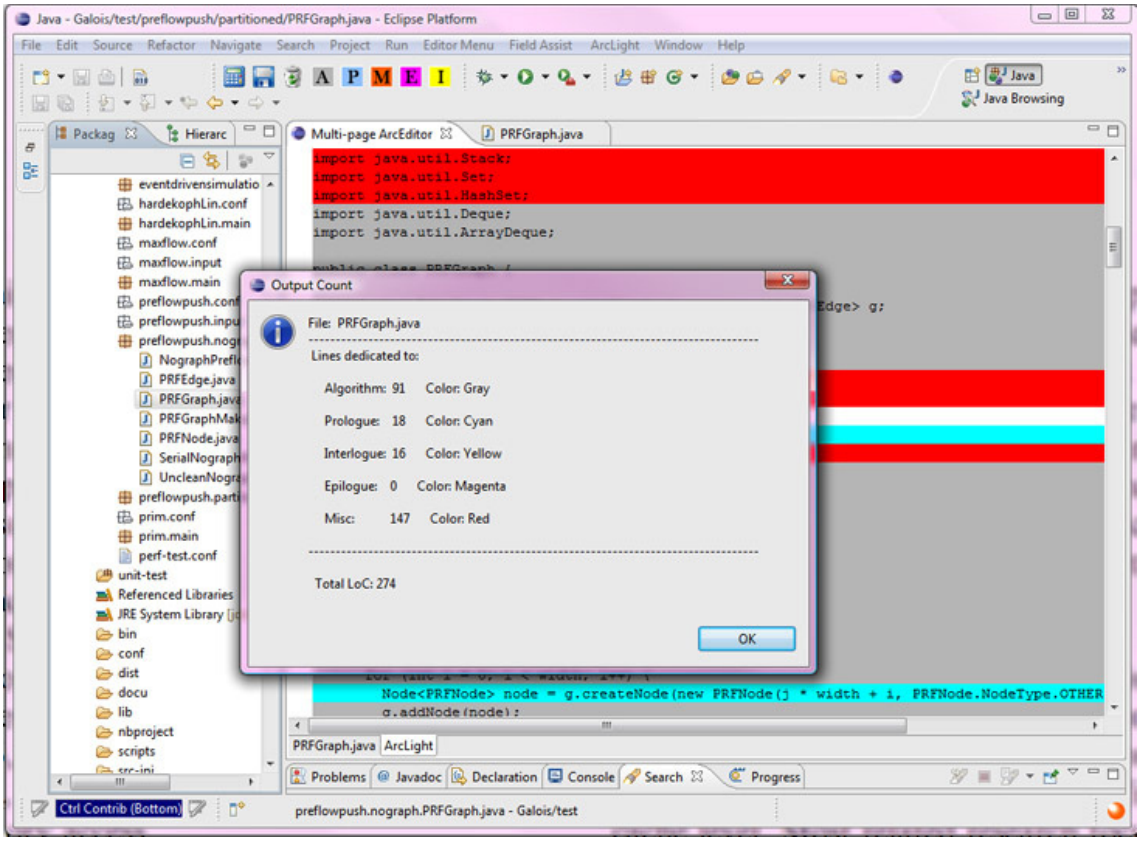


Fig. 15 – ArchLight Eclipse Plugin.

This tagging of the code allowed us to retrieve some sizing metrics (**Fig. 15**) to evaluate the percentage of code available for the identification of the patterns. This procedure was performed on four of the fourteen algorithms currently implemented using Galois. The algorithms analyzed are described in chapter 4. The results achieved are summarized in **Fig. 16.** These results were achieved by a measure of the number of *lines of source code (LoC)* per code concern, disregarding comments, empty and single character lines. A total of 2024 lines of code where analyzed.

	Algorithm	Prologue	Interlogue	Misc
Average Percentage of code	63%	2%	1%	34%
Average Percentage of code (without miscellaneous code)	95%	3%	2%	0%

Fig. 16 – Percentage of Galois code in the code of irregular algorithms

These results led us to believe that the amount of code directly related to Galois is very small indeed, on an average of 5% of the total LoC written for a given algorithm. The absolute number of LoC varies according to the complexity of the algorithm but in the implementations analyzed, this was on average a mere three to five lines of code per class. This in turn meant that the amount of available code to extract patterns was indeed limited and therefore the number of patterns able to emerge is also limited.

Another conclusion we can derive from these results is that Galois' implementation of an algorithm does not greatly affect the base algorithm's design.

6 Patterns and Pattern Languages

Over the last 30 years, the field of software development has been evolving at an accelerated rate. New and progressively more advanced techniques arise on a daily basis to the point that it is no single person can hope to grasp the existing volume of knowledge on software construction. Nowadays, the number of available software development techniques is so immense that programmers are ever more focused on small, specific areas of the software domain. Deciding on a specific methodology and development strategy with which to implement an algorithm is almost impossible and programmers often opt to use the solution they know best, even if it is not optimal. Thus, in order to reuse good software development practices and techniques, it is essential that expert programmers identify and document the best practices in their specific domain. In this context, *Software Patterns* represent well-documented template solutions that can be reused whenever a certain problem arises in a well-known context [6].

6.1 Patterns

For years, software developers had to rely on their own knowledge and intuition to understand which solutions were available and to decide which of those was the ideal solution for a specific problem. This meant that programmers had to be versed in a multitude of domains and methodologies, from different paradigms, frameworks and languages, to programming libraries, algorithms, databases, web and networks, parallel programming, compilation, hardware architecture, etc. The list is immense and the sheer amount of knowledge required to have even a broad overview of all these subjects takes years of study and dedication. Patterns help to reduce this effort.

Patterns capture formal solutions to specific problems, while maintaining a high level of abstraction. Thus, software patterns support a high-level form of reuse, which is independent from methodology, language, paradigm and architecture [11]. Using patterns, both expert and non-expert programmers can use and improve upon proven concepts and solutions to some of the most common problems in a specific domain, avoiding common pitfalls and benefiting from carefully thought design strategies.

For a pattern to be accepted by the community as representing a valid solution, the knowledge it conveys should be widely recognized as being mature and complete representations of a tangible solution to a problem that arises in a well defined context within a specific domain. Therefore, patterns must be concrete enough so as to represent valid solutions, yet their context should be relaxed enough to allow their application to a variety of problems.

The idea of describing reusable problem solutions as patterns first arose in the beginning of the 1970s in the domain of architecture and as a form of capturing solutions to common design problems on the construction of buildings and towns [48]. Alexander, the architect and author of the idea, latter coined the term *"Pattern"* to describe what he deemed to be "a perennial solution to a recurring problem within a building context, describing one of the configurations which brings life to a building."

In the software development community, patterns were first introduced by Beck and Cunningham [49] in 1987. However, the true impact of patterns for software development only became apparent when, in 1995, Gamma, Helm, Johnson, and Vlissides published a book containing 23 software *design patterns*. The book was so widely accepted that Gang of Four, i.e. the four authors, quickly became synonym with software design patterns. Design patterns represent design problems and their respective solutions and entail cooperation between classes and object. These represent only a subset of the overall set of software patterns since they do not consider computational problems such as algorithms or structural problems such as parallelism and distribution [50]. After the popularity of the Gang of Four patterns, pattern-oriented software development became a prolific area in the domain of software development. Patterns spawn multiple application domains such as object-oriented programming [7], aspect-oriented programming [8] framework design [9-10], software architecture [11-12], components [13], machine learning [14-15] and even patterns about patterns [16-17].

As the pattern community continues to grow, software developers are once again confronted with an immense amount of knowledge that is difficult to process. Patterns are swiftly becoming what they were created to prevent in the first place [51]. To prevent this, several pattern repositories are being created in the web, helping software developers access the various pattern collections and differentiate their purpose and applicability. Some repositories have a wide range of patterns: the Semantic Framework for Patterns (SFP) repository

references various software pattern collections and accounts for over 2900 patterns[1]. Others are more focused, like the Hypermedia Design Patterns Repository [52] or the Human-Computer Interaction and User Interface Design pattern repository [53]. PatternForge, a wiki for the EuroPLoP 2007 Focus Group on Pattern Repositories, lists 29 pattern repositories available on the web [2, 54].

6.2 **Pattern languages**

When patterns are considered in isolation, as single entities, software developers cannot be fully aware of how the pattern was originally intended to be composed with other patterns. Using stand-alone patterns for real-world systems frequently results in added an increase in design complexity, since single patterns cannot consider the multifaceted context of large scale software [55].

Pattern catalogues should therefore be introduced as *Pattern Languages*, which consider pattern dependences and guide *pattern-oriented software development*. Pattern languages form complete sets of patterns, such that choosing to use one pattern will eventually direct the software developer to use another related pattern. Following the sequence of pattern dependences will lead to a complete solution for a complex context.

However, there is no formally defined rule for defining pattern dependences in pattern languages. Dependences are usually introduced by a graphical mapping of dependences [56-57] or, more traditionally, through a *Related Patterns* section in the body of the pattern [6]. Graphical descriptions of the dependences between patterns are very useful to provide an overview of how patterns interact. However, they only present short non-descriptive commentaries. A Related Patterns section is more verbose but can often be interpreted in slightly different ways. Therefore, most modern pattern languages use a composition of both forms since they are complementary [58].

Independently of the recognized benefits of using pattern languages, there are still significantly more independent patterns than complete languages. Booch recently presented a study that identified 1938 patterns, collected from a set of 1884 individual patterns and only

[1] Currently the repository is being migrated to a new server. In April 2009, the repository is referenced as cataloguing a total of 2935 patterns from 234 pattern collections [58].

54 pattern languages [59]. This fact proves that there is still much work to be done in the field of pattern languages for software development.

6.3 Form and Style

There is no consensus on the formal structure of pattern description and many authors coin their own format. The main templates of pattern description relate to Alexandrian form [48], Gang of Four form [6] and Coplien form [17]. In Alexandrian patterns, the general form and style includes pattern name, context, main (problem statement, forces, solution instruction, solution sketch, solution structure and behavior), and consequences. The Gang of Four design patterns presents a structure composed of Pattern Name and Classification, Intent, Also Known As, Motivation, Applicability, Structure, Participants, Collaborations, Consequences, Implementation, Sample Code, Known Uses, and Related Patterns. Coplien Patterns are based on Alexandrian form but present patterns in a reduced format, which includes Pattern Name, Problem, Context, Forces , Solution, Rationale, Why does this pattern work? and Resulting Context.

There are several other forms for pattern description in use by the pattern community. However, in general, there are five elements that are consistent in the various formats: pattern name, problem description, context, forces and solution.

Pattern name

The name of the pattern needs to convey a sense of the purpose of the pattern and are usually used as substantives in the body of the pattern.

Problem description

This section describes the essential information about the problem the pattern proposes to solve. It is usually a small paragraph where the author states the question that conveys the problem.

Context

The context in which a pattern can be applied is of utmost importance since it allows software developers to understand if the solution this pattern represents can be applied in the context of their own problem. The context should be formally presented, so as to not overlook any

preconditions or consideration, while at the same time allowing it to be applicable to various contexts.

Forces

Forces represent constraints and decisions and are often described in pairs of opposing considerations, tradeoffs or compromises. The forces section is not standardized among the various authors.

Solution

The solution is a description of the steps proposed to solve the problem in a particular context.

(Page intentionally left blank)

7 Pattern Language

In this section, the patterns found within the Galois Framework are described as a pattern language.

The pattern language is structured so as to separate the various concerns regarding the implementation of an irregular algorithm in an amorphous data-parallel manner (section 2.1). It intends to be as general as possible, and although its main objective is to describe the patterns found in Galois, its usefulness is not specifically limited to the Galois framework. An overview of the pattern language is shown in **Fig. 17**.

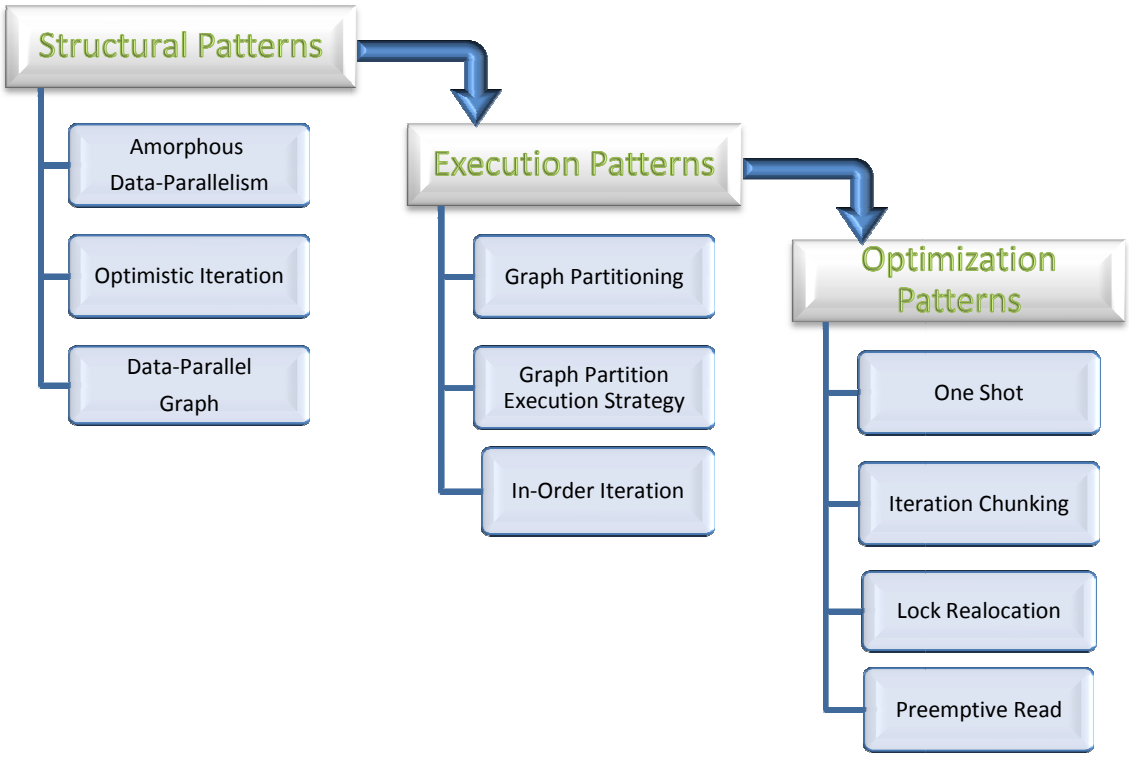


Fig. 17 – Overview of the Pattern Catalogue.

The design of the pattern language follows a hierarchical structure that represents the order in which these sets of patterns should be applied, that is, execution patterns intuitively build upon the structural patterns implementation and optimization patterns build upon execution patterns. The implementation steps are congruent with the actual considerations that the programmer must take when deciding to implement an irregular algorithm and are divided into three separate sets:

1. **S*tructural patterns*** consider how to structure irregular algorithms in terms of their algorithmic properties and data-structures and how these will be affected by optimistic parallel execution. This set of patterns is the most important of the pattern language, since the two underlying design spaces build directly upon its properties. If the programmer cannot conform to these patterns, then using patterns from this language is discouraged.
2. ***Execution patterns*** effectively take into account how the actual execution of the algorithm is handled and how to guide the algorithm to explore the maximum amount of parallelism. Not taking these patterns into consideration may lead to lower performance benchmarks.
3. ***Optimization patterns*** are designed to present the final phase of implementation and essentially focus on some optimizations that, when applied over *structural patterns*, contribute to further increase the performance of irregular parallel algorithms.

In this sense, this pattern language is meant to be applied as a sequence of steps that will eventually transform an irregular algorithm into a Galois parallel irregular application. However, there are some considerations as to the actual application of the patterns since there are relationships and dependences among them that might provide further insight to the applicability of a pattern at a given moment. The diagram of **Fig. 18** further describes these relationships.

Note that this is in fact a conservative representation of the relationships among patterns and not every relationship is explicitly represented as an arrow – the different levels of the patterns also imply dependences. Every *optimization and execution pattern* requires the application of *structural patterns* and, while their implementation is not strictly necessary, *execution patterns* should at least be considered prior to any o*ptimization.* Additional analysis

of dependences among patterns shall be referred to in the related patterns section of each pattern.

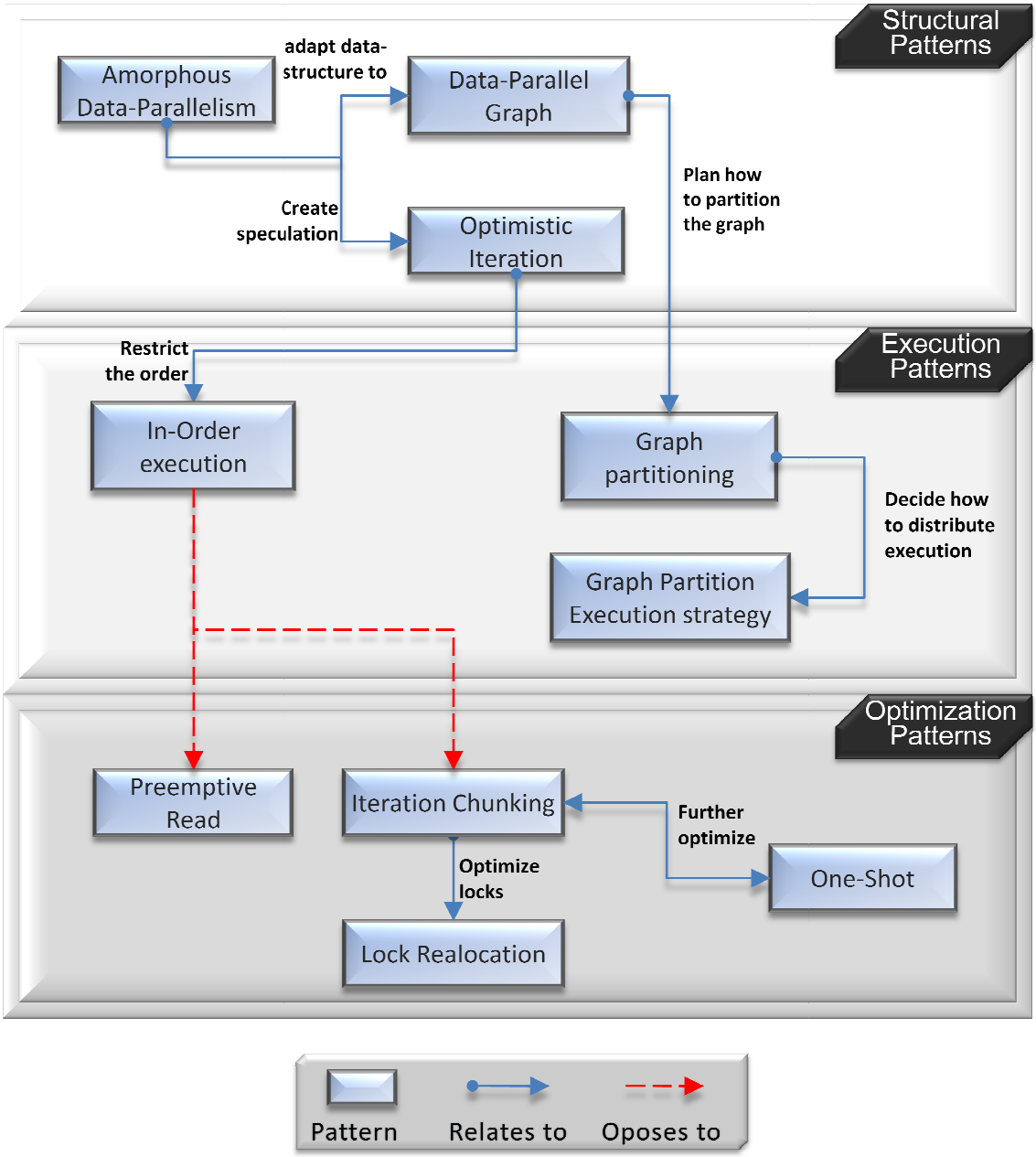


Fig. 18 – Explicit relationships among patterns.

This tight dependence between patterns is a clear indicative that this is in fact a pattern language and not a mere catalog.

The form and style used for the pattern language builds upon the pattern name- problem-description-context-forces-solution form, described in section 6.3 , and includes the Also Known As, Galois Implementation, Example, Related Patterns and Known Uses sections:

Also Know As – presents several alternative names, representing the same concept but directed at different domains or methodologies.

Galois Implementation – in this section we present some concepts of how this patter is implemented in the Galois Framework. It is complementary to both the Solution and the Example sections since it presents a case study of how the pattern can be applied.

Example – the examples section validates the pattern by using it to solve a well known problem.

Related Patterns – this section presents small textual descriptions of dependences the pattern has with other patterns in the same pattern language or with external patterns.

Known Uses – together with the example section, this section validates the pattern by demonstrating its use in the software development community.

7.1 Pattern Terminology

In order to help the reader abstract from algorithm and implementation specific jargon, we describe in this section some of the more general terms used in our pattern language:

Available parallelism – number of iterations available for concurrent execution at any single instance in time.

Data element – a *data element* is a well-identified, describable unit of data that may be indivisible or consist of a set of data items. Additionally, data elements can be individualized from the overall data set. The identification of such data elements is algorithm-specific and usually comprises on an often repeated name whose meaning is associated to the algorithmic metaphor. Using examples of the previously described algorithms, for the *Preflow-push Algorithm* (section 4.2 a data element is a node on the path from source to sink, while on *Delaunay Triangulation Algorithm* (section 4.1 the data element is a triangle on the mesh.

Data Set – a set of *data elements* usually represented as a data-structure.

Iteration – an *iteration* represents the unit of a step in the solution of an algorithmic problem. To solve an algorithm implies repetitively applying that step a finite number of times, i.e., to iterate an algorithm is to repeatedly apply the same operation over a set of data until de algorithm terminates. Moreover, often it involves using the output of an iteration as the input of its predecessor.

Work – a work element is equivalent to an iteration since when processing iterative work-list based algorithms, the element that is retrieved from a worklist represents an iteration of the algorithm.

Set of neighbors or neighborhood – represent the set of data elements will be read or written by a computation.

Processing Unit – represents either a processor core or thread and is used to abstract from the actual processing element. Therefore, our patterns can be used in both multi-core and multi-threaded environments.

7.2 Algorithm Structure Patterns

7.2.1 Amorphous Data-Parallelism

Problem

How to exploit concurrency in the presence of unpredictable data dependences?

Context

Traditional data parallelism exploits the decomposition of data-structures as a way to attain concurrent behavior. This entails dividing the data structure into independent sets and distributing them among processing units in a way that allows for the parallel application of a stream of operations.

However, when dealing with irregular algorithms, the nature of data dependences is unpredictable and dynamic and the amount of parallelism that can be achieved varies according to how the algorithm changes its data dependences. As such, the decomposition of the data-structure cannot be statically defined.

Amorphous data-parallelism is a particular form of data-parallelism that arises when the underlying data-structure has no fixed shape or size, i.e. is *amorphous*, implying that the amount of available opportunities for concurrency-free parallelism is unpredictable.

How then can we decompose an algorithm's data in a way that allows for data-parallel execution, when:

1. The occurrence and location of data accesses can only be properly estimated at runtime.
2. Concurrent computations may modify the structure of underlying data.

Forces

- **Data Granularity**
  Coarse-grained data may imply less communication but will introduce larger computational overhead and reduce the amount of available parallelism opportunities. If on the other hand the grain is fine, communications will represent the major overhead but will introduce a greater amount of available parallelism.
- **Redundancy vs. Communication**
  In a distributed environment, it can be profitable to perform redundant calculations in each of the distribution locales, instead of relying on data communication. This can introduce scalability opportunities.
- **Sequential to Parallel Traceability**
  If the pattern is well applied, there must be a simple and convenient mapping between the sequential and parallel versions of an implementation. This allows programmers to easily check the correctness of their implementation.
- Modifications to the data-structure do not create deadlock opportunities.

Solution

On irregular programs, there must be an innate knowledge of how the different parts of the program interact and what part the data plays in the overall solution design. This is the basic strategy for the exploitation of data parallelism. In this context however, one must also consider how the concurrent behavior will operate over the data and how to ensure the independence of computations in the overall parallelization strategy.

As such, the general solution for this problem entails being able to, at each iteration:

1. Identify the independent sets of data able to be executed in parallel
2. Decide which shared data elements need to be locked to avoid concurrent access
3. Ensure that the computational cost of independent sets remain balanced

Also, a decomposition based on amorphous data-parallelism must ensure that:

- Data dependent computations drive parallelism.
- Computations are performed in a way that introduces opportunities for independent parallel execution over the data.

The general solution of this pattern is comprised of the following steps:

Step 1 - Determine the type of algorithm operator

Following the characteristics described in *Categorization of irregular algorithms2.2* (section 2.3) the operator of an algorithm can be one of three types: *Morph, Local Computation* and *Reader*.

The semantics of the operator is defined by degree of influence:

- Morph algorithms have at least one morph operator.
- Local Computation algorithms have no morph operator and at least one local computation.
- Reader algorithms have strictly reader operators.

Step 2 - Define a valid data-parallel decomposition based on the concept of basic data element

The basic data element represents the smallest independent set of data around which the parallelism will be driven. Defining this will allow the programmer to consider how to apply locking mechanisms to ensure concurrency. Together with the operator, the definition of data element allows us to understand of how each iteration changes the structure of data and how to identify the set of independent data elements.

Step 3 - Express computations in terms of the data-structure elements.

The programmer must choose how the data-structure will be iterated and reify an abstraction of the data for the computation as a call to DataStructure.get(index) or some similar instruction. This step is highly influenced by the choice of parallel programming language.

Step 4 - Repeatedly apply the computation algorithm to each data block.

This means not only iterating over the data-structure and applying the computation but also checking for the constraints of amorphous parallelism:

```
foreach element in dataStructure atomically do
      needed_data_blocks = // identify neighbors
      lock needed_data_blocks;
      compute(dataStructure.get(needed_data_blocks));
      unlock needed_data_blocks;
endForeach
```

In this case, the atomicity of the operation and locking mechanisms are meant to restrict the ways in which concurrent access can invalidate the computation.

Galois Implementation

In the Galois implementation of this pattern the main consideration are:

- The main loops of the algorithm must be refactored to use the Galois foreach loop;
- The Galois foreach loop uses a worklist as the iteratable data-structure (section 3.2). Thus the points that compose the mesh must be added to the worklist prior to any computation;
- Locking is abstract, i.e. it is implicitly handled by the Galois Runtime System (section 0).

Listing 8 (L8) and **Listing 9** (L9), present the main loop of the serial and parallel versions of the Delaunay Triangulation algorithm implemented in Galois.

```
1  Stack<Tuple> worklist = new Stack<Tuple>();
2  //initialization of the elements
3  for (Tuple tuple : worklist) {
4     //split the triangle that the tuple falls in into
        three triangles
5     //Check list of edges and flip if necessary
6  }
```

Listing 8 – Sequential Delauney Triangulation

```
1  UnorderedWorklist<Tuple> worklist =
                GaloisWorklistFactory.makeDefaultWorklist();
2  foreach (Tuple tuple : worklist) {
3      //initialization of the elements
4      //split the triangle that the tuple falls in into
         three triangles
5      //Check list of edges and flip if necessary
6  }
```

Listing 9 – Galois' Parallel Delauney Triangulation

The differences in the code comply with the considerations described in the beginning of this section:

- The sequential version uses a simple stack as a worklist (L8-line 1), while the parallel version uses one of the worklists available via the GaloisWorklistFactory (L9-line 1).
- The sequential version's main loop uses an ordinary for loop(L8-line 3), while the parallel version uses Galois's foreach loop (L9-line 2) as the main driver for parallelism.
- The elements locally needed by the algorithm need to be initialized within each thread and so, in the parallel version, the code must be moved inside the loop (L9-line 3).
- Galois tries to minimize the number of changes that need to be performed to parallelize an irregular algorithm. On the majority of cases, no additional changes need to be made, which means that parallelization of irregular algorithms can be achieved without much effort [2, 34].

Example

Using the Delaunay Triangulation example (section 4.1), the underlying algorithm problem can be parallelized in an amorphous data-parallel way by considering each triangle as the data block that drives parallelism. The repeated application of an activity to the various triangles exposes the parallelism inherent to the algorithm.

As for the constraints of amorphous data parallelism:

1. **Data may be subject to concurrent access.**

   Two different iterations can try to access the same point. This is more obvious when checking the Delaunay Property since the neighborhood of a triangle is extended to encompass the points that can be influenced by the flipping operation.

2. **The occurrence and location of data accesses can only be properly estimated at runtime.**

   The actual triangle mesh is created dynamically by each iteration of the algorithm, meaning that the validity of the Delaunay Property for each triangle can only be checked after the actual triangulation. This also means that flipping is runtime dependent.

   Furthermore, since points are randomly chosen, the triangle mesh that will be generated cannot be estimated statically.

3. **Concurrent computations may modify the structure of underlying data.**

   Flipping modifies the edges that compose the mesh. Another way in which computations influence the structure is when a point is added to the mesh and that point invalidates the Delaunay Property for a number of triangles that must be re-triangulated.

In the Delaunay Triangulation example (section 4.1), these three steps would consist in:

1. Defining that a triangle will be the basic data block. This means that every operation will be computed over a triangle and the point it encloses.
2. Express the triangulation in terms of the data blocks. This means retrieving the surrounding triangle for each new point to be triangulated:

```
mesh.getSurroundingTriangle(point);
```

3. The computation would be something in the terms:

```
foreach point in the graph atomically do
      lock surrounding_triangle;
      trianglulate(surrounding_triangle, point);
      unlock surrounding_triangle;
endForeach
```

Related Patterns

- ***Data Decomposition***

  Amorphous Data-Parallelism can be considered as a more specific form of *Data Decomposition* [56].

- ***Loop Parallelism***

  As the name implies *Loop Parallelism* [56] helps uncover parallelism based in loops, which is a central part of *Amorphous Data-Parallelism*.

Known Uses

This pattern was first described by Kulkarni [34], although, to the best of our knowledge, we are the first to call it a pattern. Recently, Lublinerman *et al* [60] presented Chorus, a high-level parallel programming model for irregular applications which uses the concept of *amorphous data-parallelism*.

7.2.2 Optimistic Iteration

Also Known As

Data-Driven Speculation, Speculative Execution, Optimistic Execution

Problem

How to efficiently parallelize an algorithm that presents an amorphous data-parallel structure?

Context

This pattern implies Amorphous Data-Parallelism.

Recall that amorphous data-parallelism arises on worklist-based irregular algorithms implemented over dynamic data-structures. Execution of these algorithms is governed by the many and dynamically changing data dependences between iterations. When considering how to efficiently parallelize such algorithms, a more traditional approach using locks to synchronize concurrent accesses to data is possible, but would undoubtedly reduce the amount of available parallelism. Other alternatives include using static analysis techniques, like points-to and shape analysis, or semi-static approaches, based on the inspector-executor model, to try and uncover an higher amount of potentially concurrently executed code. However, both these models fail to uncover the full set of potential parallelism, since static analysis techniques only check data dependences at compile time and semi-static approaches do not acknowledge dynamic dependence changes in data-structures.

To overcome the dependence chain under these conditions, programmers must take into account the advantages of speculative or optimistic parallelization techniques [34]. For this specific case, speculative execution of *Amorphous Data-Parallelism* implies being able to execute parts of the code without complete knowledge of the data dependences.

Forces

- **Implementation Cost vs. Benefit**

  Implementing an optimistic execution technique from scratch can be costly. The main disadvantage of these techniques lies is the complexity of handling miss-speculation problems, such as state saving and rollback actions. These can be quite challenging and if not done properly can increase the memory and computational cost of an algorithm to the point that there is no added benefit in using *Optimistic Iteration*.

- **Available Parallelism vs. Number of Conflicts**

  While finer-grain computations can lead to a greater amount of available parallelism, it will also increase the likelihood of conflicts. Therefore, it the programmer should consider how many independent computations can occur at the same time and define the size of the grain accordingly – not too fine and not too coarse.

- **Grain of Parallelism vs. Cost of Locking**

  If the cost of locking is equivalent whether a computation is fine or coarse grain, then executing many fine-grained computations might be worse than executing a single coarser one.

- **Grain of Parallelism vs. Cost of Miss-Speculation**

  The cost of miss-speculation can be considered as the sum of the cost of corrective action with the cost of re-executing the work, added with the cost of acquiring and releasing locks for both the conflicted and re-executed iteration. Therefore, the cost of miss-speculation increases as the grain coarsens, as does the amount of wasted work.

Solution

The idea behind *Optimistic Iteration* is to execute an algorithm in parallel while assuming that data dependences are never violated, that is, that there is no concurrency in the access to data elements. This does not mean that data is truly independent but merely that if the system detects that a dependence violation occurred, it will take appropriate corrective actions. When no violations are detected, the results of iterations can be committed and the resulting data elements are added to the data dependence set.

Optimistic Iteration techniques are widely used in the parallel programming community and there are several different strategies for the implementation of speculative mechanisms. It is unfeasible to describe all the specifics of the different techniques in detail. For this reason, we have selected what we consider to be the main application-independent focal points of *Optimistic Iteration* and refer further details on the various techniques to the know uses section.

The following steps describe how to speculatively execute an algorithm in an Amorphous Data-Parallelism way:

Step 1 - Determine the type of algorithm operator

Following the characteristics described in *Categorization of irregular algorithms2.2* (section *2.2*) the operator of an algorithm can be one of three types: *Morph, Local Computation* and *Reader*. Strictly-reader algorithms don't have much to gain from *Optimistic Iteration* since the structure of data dependences never changes.

Step 2 - Build data dependence graph

Optimistic Iteration uses the speculative execution of iterations as a way to break the highly coupled dependence chain around data elements. This means that, in order to create a valid mapping from data to iterations, the programmer needs to build the data dependence set for the specific algorithm under consideration. To this end, *Data-Parallel Graph* constitutes a good and useful abstraction when it comes to parallel processing. A graph can be seen as if composed of computational nodes connected by edges encoding computational dependences. This means that where we have a data node, we can assume that there is a corresponding iteration. In addition, to every edge connecting two nodes, and therefore representing data dependences, we can assume that there is a corresponding edge between iterations that represents computational data dependences, that is, data outputted from one iteration is inputted in another. This abstraction allows us to consider the various iterations of the algorithm as a traversal of data dependences.

Step 3 - Anticipate special ordering restrictions between iterations

The programmer must consider just how strict is the data dependence between the different iterations. If iterations must be committed in a sequential-like order, then *In-order Iteration* applies.

Step 4 - Predict the set of neighbors of each iteration

This is the most important and difficult step. For most of the irregular algorithms, the neighborhood can be predicted with a certain degree of accuracy. This prediction involves understanding which data elements will be read or written on each iteration. On *local computation algorithms*, the neighborhood can be approximated in a straightforward manner since the structure of data dependences never changes. In matrix based algorithms, for example, the values of the matrix might change with every iteration but its structure remains the same. *Morph algorithms*, on the other hand are harder to predict since every iteration might change the structure of data dependences. This means that while we can predict that a neighborhood is a set A of data elements, another parallel executing iteration might add a new element to the structure (say element b) which will in fact increase the neighborhood A to $A \cup \{b\}$. In this case, the neighborhood cannot be properly estimated and we are clearly in a situation where *Optimistic Iteration* is the best option.

Step 5 - Introduce locking mechanisms

The programmer must lock every neighboring data element with whichever locking mechanisms the implementation language or framework provides. Although optimistically assuming that there will be no concurrent access to data elements, it would be foolish not to lock the elements we are currently accessing. Locks are only released immediately prior to committing the iteration. This adds atomicity to an iteration, in the sense that the data-structure always maintains a consistent state.

An additional consideration towards locking mechanisms reinforces the fact that these locks should not be all restrictive. That is, some operations should be allowed to perform concurrently while others require exclusive access to data.

Step 6 - Consider how to handle miss-speculation and rollback operations

In contrast with many traditional approaches to concurrent execution, optimistic execution does not actively avoid conflicting data accesses. Locks exist only to guarantee that nothing affects the data that an iteration is currently accessing. when an iteration tries to change the data elements whose lock is hold by another, a conflict occurs. The programmer must take careful consideration so as to ensure that no conflict goes unnoticed by the system, otherwise there is no guarantee towards the correctness of the end result. When a conflict is detected, optimistic methods must be able to recover from this, without deadlocking or waiting for the locks to be released. Recovering from an illegal access requires that the iteration be reset to its initial state. There is a broad variety of methods and variations to provide this type of operation [34, 61-62]. The main methods of performing rollback are described next:

- **Lazy update –** changes are performed in cache and are only moved to main memory after the iteration commits successfully. This is identical to shadow copy, where all operations are performed on a copy of the data element that then replaces the original, if the iteration commits successfully.
- **Undo operations –** all operations are stored in an undo log. Rolling back an iteration is just a matter of reversing all modifications to the data-structure in a last-in first-out manner.

- **Snapshot –** prior to any change, a snapshot of the data is saved and all changes are performed on the original data-structure. In case of a rollback, the snapshot is recovered and replaces the modified data, restoring it to its original state.

After rollback, the iteration either is allowed to try again immediately or waits to be processed later.

Step 7 - Release all locks

Whether the iteration is able to commit or has to rollback, the last step is to release all locks that the iteration acquired and proceed to the next iteration.

Galois Implementation

As described in chapter 3, Galois is an object-based optimistic parallelization framework for irregular algorithms and therefore, has built-in structures that support optimistic execution. These are provided via the three main aspects of the framework:

Programming Model

The programming model requires the programmer to represent the main loop of the algorithm as a set iterator. This unbounded set iterator will be used as a worklist for the algorithm, in fact helping introduce optimistic parallelism by means of the runtime system. By unbounded we mean that the size of the set may vary throughout the execution, as more elements are removed or added.

To help understand how the programmer manipulates data, it suffices to say that direct memory manipulation is not allowed. All accesses to data are performed via object and method invocation.

Library Classes

Galois' library provides the data-structures and shared objects implementation, specifying special properties that allow the runtime system to understand how these can be used in an optimistic way. It is the responsibility of the library to ensure that set iterators retain sequential semantics while being optimistically executed. Therefore, iterations must remain:

Consistent – any update to shared objects is atomic, that is, methods that access shared objects must ensure mutual exclusion on updates.

***Independent* –** parallel execution must follow some possible sequential scheduling. This means that executing iterations can only see committed iterations and therefore "believe" to be following some sequential order.

***Atomic* –** the state of iterations must have *all or nothing* semantics, meaning that either an iteration successfully commits or shared objects will remain as if the iteration never started.

Library classes are also responsible for deciding which operations represent access violations and which do not. This is introduced through the property of *Semantic commutativity*, which states that if an ideal schedule of operations exists, then there are some operations over locked data elements that don't need to respect mutual exclusion. This property guarantees that if two methods commute, the execution of one will not change the result of the other. Furthermore, the library also provides rollback functionality, ensured by *inverse method* semantics. Each method that changes data, either by changing the data-structure or by updating a data element, has an inverse method that undoes the action of the former. A short example of the specification of semantic commutativity is represented in **Listing 10**.

```
  [method]
void add(Element x);

  [commutes]
   add(y)            {y != x}w2
   remove(y)         {y != x}
   contains(y)       {y != x}

  [inverse]
    remove(x)
```

Listing 10 – Commutativity and inverse of a Galois library method.

Runtime System

The runtime system is responsible for both checking commutativity constraints and enforcing rollback operations, in essence ensuring that iterations behave according to the rules of *Optimistic Iteration*.

The runtime is composed of a *scheduler*, responsible for fetching work from the set iterators and creating optimistic parallel iterations, and an *arbitrator*, which while executing an algorithm, and before each method invocation, checks the method's commutativity against all other executing methods. If the method commutes, there is no race condition and the iteration can continue. Otherwise, the iteration to whom the method belonged is rolled back. In addition, to prevent concurrent rollbacks when a method is checked for *commutativity,* it is also checked for its inverse.

Example

Picking up the example of Delaunay Triangulation (section 4.1 we can elaborate on the previous implementation and create a rough optimistic version of the algorithm. In the example shown in **Listing 11**, a worker thread starts an unbounded while loop (line 5) and asks for a new iteration from the scheduler (line 7). Each iteration then creates a new triangulation and, if that triangulation is invalid, it is added to the worklist (line 12-13) and the thread iterates again to correct the problem. If there is some conflict between iterations, an

```
1  Graph graph;
2  Worker worker;          //worker thread
3  Scheduler scheduler;
4
5  while (true){
6     try{
7        Iteration it = scheduler.newIteration(worker);
8        do {
9           scheduler.nextElement(it);
10          Cavity cav = triangulateOrFlip(graph,it);
11          graph.replaceSubgraph(it, cav);
12          if(cav.isInvalid())
13             scheduler.addWork(it, cav);
14
15       } while(it.workLeft());
16
17       scheduler.commitIteration(it);
18
19    }catch (violationException ve)
20       //do nothing, graph is only updated on commit
21    }
```

Listing 11 – Optimistic implementation of Delauney Triangulation.

exception is thrown (line 19), otherwise the iteration is allowed to commit (line 17). This example is similar to a Galois implementation but since in Galois the set iterators are controlled by the scheduler, who is responsible for providing iterations to threads and to keep supplying work, by replacing the foreach loop by an unbounded while loop we instead create a more transparent version of what Galois usually does behind the scenes.

Many other different optimistic parallel implementations of Delaunay triangulation have been proposed by the parallel programming community [63-66].

Related Patterns

- ***Speculation***

  Speculation [67] represents a higher level description of a solution to the same problem.

- ***Amorphous Data-Parallelism***

  The best way to handle *Amorphous Data-Parallelism* is by *Optimistic Iteration*.

- ***Data-Parallel Graph***

  The graph data-structure provides an appropriate data-structure for *Optimistic Iteration*, since it provides an ideal abstraction for the dependence graph.

- ***In-order Iteration***

  If iterations have a restrict scheduling order, then the *In-order* pattern applies.

Known uses

The first examples of optimistic parallelization were introduced in the 70s as a form of branch speculation [68-69]. Years later, in 1985, Jefferson presented one of the most well known optimistic methods: the *Time Warp mechanism* [70]. This mechanism implemented a method for transparently synchronize discrete-event simulation in distributed systems. Other known optimistic techniques relate to loop speculation [71-72]. Recently hardware techniques have enabled optimistically created parallel threads by tracking dependences by monitoring memory accesses made by loop iterations [73-77]. This technique, known either as *Thread level speculation* or *Speculative Multithreading,* has proven to be quite useful to

optimistically parallelize many applications and has been introduced in a considerable number of parallelization architectures [78-82].

7.2.3 Data-Parallel Graph

Problem

How does a graph abstraction influence the opportunities for *Amorphous Data-parallelism* and the structure of the algorithm?

Context

On implementing an algorithm, much of the effort is spent on deciding what is the best underlying data-structure on which to represent our data and what are the characteristics that make it valuable on a concurrent environment.

In this context, we present a list of some of the reasons why graphs should be used:

1. Graphs are a generally used and accepted metaphor for describing structure and behavior. Examples of this can be as varied as state machines, flowcharts, UML diagrams, BPMN diagrams, EBNF diagrams, circuits, etc.

2. Graph nodes and edges can be associated with a variety of meanings and be of varying complexities.

3. Graphs can be used to represent virtually every data-structure used in computation. The most common examples are:
   Trees – are a form of specialized bipartite, connected, acyclic and undirected graphs with one of its element distinguished as the root element [83]. Trees have many specialized forms (like the Binary-tree, Red-black tree, B-tree, AVL tree, etc) and can be used to represent other structures like hashtables and heaps.

   Lists – represent *path graphs* [83], acyclic graphs where every node is connected to at most 2 other nodes. Lists can be used to represent stacks, pipes and queues.

   Grids – are special distance regular graphs that can be represented in two dimensional space. Grids can be easily transformed into *cubes* (in three dimensional space) or

hypercubes (above the three dimensional space). Grids can also be used to represent N-dimensional matrices.

4. Graphs represent structure and introduce constraints and properties such as hierarchy, connectivity, edge direction and weight, as defined in Graph Theory [83].
5. There is a large number of algorithms for graph traversal and search. The list includes *Depth-First* and *Breadth-First* traversal, *Iterative In-Order* and *Post-Order* [84], Dijkstra's *Shortest Path* [85] and Kruskal and Prim's algorithm [37], just to name a few.
6. Graphs can be reconfigured with little or no effort, simply by loosening or tightening the connectivity constraints.
7. Complex graphs are composed of sub-graphs with similar structural properties. This allows for additional opportunities for divide-and-conquer strategies.

Aside from the advantages stated above, programmers should take into consideration whether the graph abstraction actually benefits the implementation of the algorithm. Some data-structures, like trees and lists, are just as mature data-structures as graphs and are more attuned to some problems than others. Nonetheless, a Graph abstraction remains a perfectly good option.

Data Parallel Graph is focused on *amorphous data-parallel* graph algorithms, *i.e.*, graph algorithms that have an inherent amorphous data-parallel structure. Thus, if the underlying data in this algorithm is an irregular, pointer based data-structure, then, by all the reasons described above, a graph is the ideal choice.

The focus of this pattern is not to provide specific implementation solutions, merely to allow us to understand how graph characteristics influence irregular problems.

Forces

- ***Specific vs. Reusable Implementation***

  A more specific graph implementation can provide additional performance to the algorithm but will make it inherently more difficult to implement and will hamper reusability. One must weigh the cost of implementation against the expected benefits.

This force can also represent the decision of implementing a graph or using an available graph library.

- ***Update Cost vs. Performance***

  There must be a careful balance between the cost of dynamically updating the graph structure and the performance of the algorithm. If updates are computationally expensive, then performance will be directly impacted in a negative way.

- ***Optimization vs. Portability***

  If the data-structure tailored to a specific hardware, then performance will be greatly optimized but portability will be reduced by a similar proportion. This also reduces the chance of reproducing highly optimized benchmarks.

Solution

The general instantiation of *Data-Parallel Graph* requires the following steps in order to be accomplished:

Step 1 - Identify algorithm-specific graph characteristics

A graph data-structure can have several different characteristics, which can be sorted in three distinct classes:

Edge characteristics

Direction: By default, an edge between two nodes is considered *bidirectional* or *undirected*. This means that there is a reciprocal relation between the connecting nodes and the graph can be traversed in any direction. However, there are some instances where edges can be one-way, that is, traversing is restricted to a specific direction. In this case, the edges are said to be *directed*. An undirected graph can be represented by a directed graph where every node is connected to its neighbors by two directed edges. An example of a directed graph is a street map, since some streets are one-way and others are two-way, while a social network represents an undirected graph.

Weight: Edges can have weights, that is, there can be a cost associated with traversing a given edge. For instance, given a map of cities modeled as a graph where every edge has a cost associated with the distance between those same cities, we

could use Dijkstra's Shortest Path Algorithm [85] to discover the shortest path between two cities.

These characteristics are completely orthogonal and we can have, for a given graph, any combination of these two characteristics.

Node characteristics

Label: A node can have a label that distinguishes it from all other nodes. This is the case of the root node in trees or the source and sink nodes in the *Preflow-push Algorithm* (see section 4.2)

Value: nodes can have values that provide some contextual reference to the algorithm in question. In the case of the *Preflow-push Algorithm* (see section 4.2), nodes have a value that indicates their height in relation to other nodes. Other examples are the case of boolean values that indicate whether a node has been visited before or color values, typical of graph coloring algorithms.

These characteristics are completely orthogonal and we can have, for a given graph, any combination of these two characteristics.

Structural characteristics

Structural characteristics of graphs infer a sense of how data is organized and help realize how special structural attributes are to be handled.

Completeness: If every node is connected to every other node, then we say the graph is *complete*. Complete graphs are difficult to handle because they cannot be efficiently partitioned due to the absence of sub-graphs.

Independence: A node is independent or isolated if it has no edges connecting it to other elements in the graph. A set is independent if it constitutes a sub-graph that is not connected by any edge to the main graph. This means that the programmer must consider this characteristic when designing traversing and partitioning strategies for the algorithm.

Connectivity: If for every two distinct nodes there is a path connecting them, then we say that the graph is connected. This characteristic influences the amount of independence present in the graph.

Cycles: A cycle exists if starting from a given node, exists a path through the graph that leads back to that same node. Most of the graphs contain various cycles and this important characteristic means that the programmer must make special attention so that the algorithm doesn't get caught in an endless loop around a cycle.

Self-loops: A self loop happens when a node has an edge connecting to itself. This is a special case of the cycles characteristic since the algorithm can be caught in a closed loop, never leaving the same node. Self-loops must also be taken into account when partitioning the graph so that there is no node duplication.

Step 2 - Define the graph data-structure

The vast majority of programming languages don't provide built-in graph data-structures. However, there are a few libraries available. This is due to the fact that a generic graph library can be quite complex and can be implemented in n-number of ways (typically as adjacency lists or matrices but there are some purely object-oriented implementations available).

On deciding which implementation of graph data-structure to use, the programmer must take into account the following two factors:

Reusability factor

On choosing or implementing a graph data-structure one must take care to identify the nature and reusability aspects of the problem at hand. If the problem is small and there is little probability that a full fledge graph data-structure will be needed, then an implementation using an adjacency list or matrix is a good alternative. This type of blunt implementation is ideal when the cost of learning how to use a third-party library or of implementing a more generic and complex graph data-structure is considerably higher than the cost of implementing the overall algorithm.

Optimization factor

More than the cost of learning how to use a graph library, the programmer must take care to consider if and how the algorithm can be optimized and how this optimization can be achieved with a wide-spectrum graph library.

If the algorithm is intended to be run on a specific computational environment and is expected to achieve the utmost performance in that said environment, then the data-structure needs to be closely attuned to the underlying hardware configuration or operation system. This means that the data-structure should be designed with these specific characteristics and trade-offs in mind. For instance, a third-party graph library doesn't have many considerations for partitioning concerns.

On the other hand, if an ideal performance can be achieved by fine-tuning the algorithm instead of the data-structure, then probably the learning curve of using a third-party graph library has a lower cost than implementing a brand new data-structure.

Step 3 - Determining how the algorithm traverses the data-structure

This is important in this context because in the case of parallel implementations of algorithms, the graph traversal is what drives parallelism. That is, is through the traversal of the elements of the graph and by performing the needed computations that the algorithm progresses. The traversal strategy is also very dependent on if and how data is partitioned.

Step 4 - Determine how amorphous data-parallel computations can be composed

At this point, it is necessary to identify how to efficiently parallelize and partition the graph data-structure. This implies the application of the *Amorphous Data-Parallelism pattern* and the *Graph pattern*.

Galois Implementation

The Galois framework supplies a few graph-based structures designed to support the aforementioned graph characteristics. These data-structures are implemented around a Graph interface, providing support for directed and undirected graphs, as well as complex, simple and indexed edges. All Galois graph classes implement the interfaces seen in **Fig. 19**.

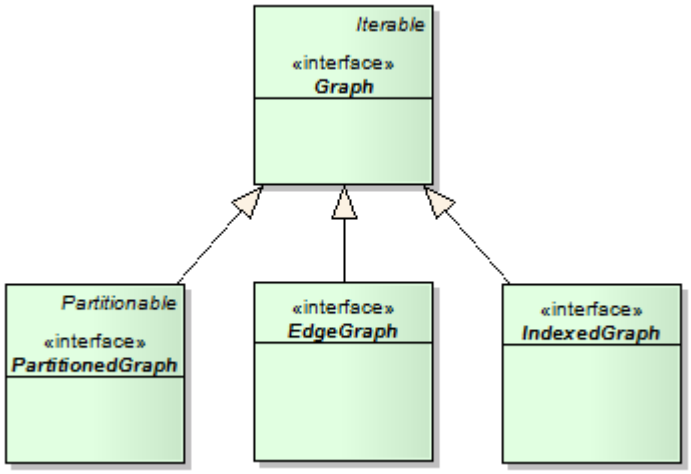


Fig. 19 – Hierarchical interface model of Galois' Data-structures.

In addition to the graph classes provided in the library, these classes can be subclasses in order to support a more algorithm-specific graph implementation. The *Preflow-Push algorithm* (section 4.2) is one such cases.

The *Preflow-Push algorithm* needed three extra specific structures implemented over the supplied data types and structures:

PRFEdge – Represents the information contained in an edge of the graph. In this case, it adds the capacity constraints and adds direction properties to an edge (source-destination). This class is wrapped by the Edge interface which provides all default edge operations.

PRFGraph – Represents an extension to the EdgeGraph class, which it wraps. Moreover, it provides all the methods for initializing the algorithm as well as the Push and Relabel operations.

PRFNode – Adds algorithm specific node information, such as height, excess, id and identifies the type of node (Source, Sink or Other). This class is wrapped by the Node interface, which provides all default node operations.

Example

As previously stated, using graphs as primary data-structures usually brings some useful advantages. This is the case of the *Sparse Cholesky Factorization Algorithm* introduced in

section 4.3 . The traditional Cholesky algorithms are implemented with a matrix-like data-structure. In this case however, since the matrix is intended to be sparse, that is, the majority of its elements are zero, the matrix can be mapped into a graph without compromising efficiency. **Fig. 20** shows two such mappings.

In this example, a matrix (**Fig. 20-a**) is mapped to a graph representation (**Fig. 20-b**) where r nodes represent rows and c nodes represent columns. Edges between the nodes map the actual values presented in the matrix. Another, more efficient mapping (**Fig. 20-c**) is obtained by creating one node per index value. This means that for an NxN matrix, there will be only N nodes. In this case, the main diagonal is represented as self-edges, while other edges are duplicated due to the matrix's symmetry. The mapping from matrix (a) to graph (c), from **Fig. 20**, can be accomplished by the code in **Listing 12**.

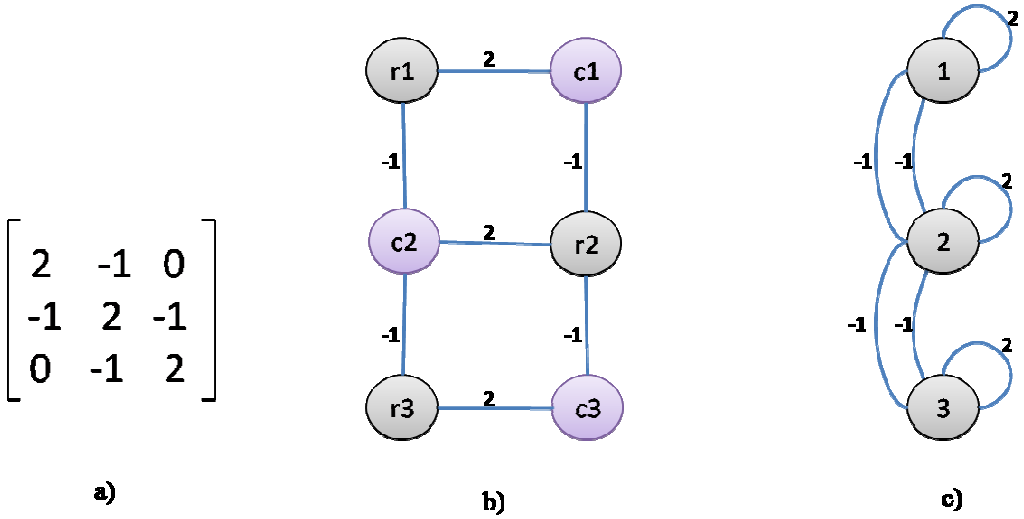


Fig. 20 – Graph representations of a sparse symmetrical matrix.

After this point, the programmer should adapt the algorithm to use this new graph representation of the sparse matrix.

```
1  Graph g;
2  Matrix [rows] [columns] m;
3  for ( int col in columns){
4     for( int row in rows : row <= col){
5        if( m [row] [col] != 0){
6           Node ncol = g.addNode(col);
7           Node nrow = g.addNode(row);
8           //create edge and add its value
9           g.addEdge(nrow,ncol,m [row] [col]);
10       }
11    }
12 }
```

Listing 12 – Matrix to Graph Transformation.

Related Patterns

- ***Amorphous Data-Parallelism***

  This pattern is used to provide an underlying graph representation to the data required by the *Amorphous Data Parallelism pattern*.

- ***Graph***

  The graph data-structure influences partitioning which in turn will influence parallelization opportunities.

Known Uses

The RKPianGraphSort [86] is a rather recent approach to the problem of sorting a set of records in some pre-defined order. Sorting algorithms [87] are a rather well know and much studied set of irregular algorithms, with irregular data accesses but using non-dynamic(vectors) or semi-dynamic(lists) data-structures. This algorithm instead uses a graph-based sorting technique that shows a non-Galois oriented application of this pattern. Also, since the algorithm does not present gains in performance when compared to previous algorithms, we believe that a Galois amorphous data-parallel implementation of this algorithm would provide substantial improvements to the base performance.

7.3 Algorithm Execution Patterns

7.3.1 In-order Iteration

Also Known As

Ordered execution

Problem

How to find available amorphous data-parallelism when tightly inter-dependent iterations constrain execution to a sequential iteration order?

Context

This pattern implies *Optimistic Iteration*.

To the majority of irregular algorithms, the order in which iterations are processes doesn't constrain the actual outcome. To some, the end result is the same in whichever order the iterations are processed. This is an example of non-deterministic order and is the case of the *Preflow-push Algorithm* (section 4.2), which always finds the maximum flow, independently on the order in which nodes are processed. In other irregular algorithms, although the order of iteration indeed changes the output, the correctness of the algorithm is maintained. The *Delaunay Triangulation Algorithm* (section 4.1) is a clear example of this. Different orderings might produce different meshes, but the output will always be a mesh on which every triangle respects the Delaunay property.

There are however, some algorithms in which the order in which the iterations progress not only influences the end result but is the only order in which we can ensure correctness. This is the case of *Event-driven simulation* [88], where events must be processed in global time order or *Kruskal's minimum spanning tree* (section 4.4 where edges must be processes by increasing weight.

When dealing with optimistic parallelization of irregular algorithms, there is a good chance that the programmer will eventually be confronted with a restrictive ordering of execution that in theory would invalidate the advantages of speculation. There are two ways in which ordering can be enforced:

- Iterations depend on data previously computed in other iterations.

- Iterations must follow data properties that enforce ordering constraints, like alphabetical or numerical order, for example.

Matching the execution order of iterations to this sequential order can be achieved statically. The problem is how to extracting Amorphous Data-Parallelism with Optimistic Iteration, when executing speculatively will in the majority of cases lead to conflicting accesses to data and to wasted work?

Forces

- ***Amount of constrain vs. Benefit***

  If ordering constrains only a very small set of iterations, then probably the cost of introducing In-order Iteration doesn't cover the benefits in performance.

- ***Order of rollback***

  The order of rollback of conflicting iterations could lead to deadlocks. If a higher priority iteration keeps rolling back due to conflicts with a lesser priority iteration, the algorithm would stop progressing and eventually might not terminate. A timeout mechanism could be an efficient way to check for priority errors.

- ***Size of data set***

  The size of the data set influences the distribution of iterations and therefore, the bigger the data set, the more opportunity for independent execution exists.

Solution

The solution passes for trying to find a way to extract a usefull amount of Amorphous Data-Parallelism while not disregarding the complexity of the ordering constraints. The steps to achieve this are described next:

Step 1 - Check for partial ordering

The majority of irregular algorithms enforce only partial ordering, that is, only relatively small sets of iterations have to respect ordering constraints. Independence between constrained sets is nevertheless possible. To illustrate this, let us consider the case where two iterations, A and B, are geometrically distant in that they don't share the same data elements. Nevertheless, some ordering is enforced – say alphabetical ordering – meaning that iteration

A would always have to be executed before iteration B. Between these two iterations there is no available optimistic parallelism because executing B before A would lead to a conflict. However, this represents only a partial ordering. There is always a possibility that two A iterations could be executed concurrently. The same concept can be applied in minimum spanning tree algorithms – usually more than one edge has the same or approximate weight – or event based algorithms with logic clocks – *Lamport clocks* [89] have causal order of events, yet a global ordering is only enforced for events that trigger actions on different processes. Same process events have only to comply with local order and can occur concurrently with other local order events on other processing units.

If the amount of iterations able to execute concurrently is considerably high, then there might be no need to further refine the implementation to better explore *Optimistic Iteration.* The amount that will be required for efficient performance is very algorithm dependent and therefore requires experimentation in order to estimate.

Step 2 - Consider committal order

If in fact, there isn't enough available parallelism and performance is constrained, another solution is to consider that when algorithms have partial ordering constraints, that order needs only be enforced when iterations commit. The state that is observed by the system must remain consistent at all times, but consistency is only ensured after committal. When iterations are executing speculatively the state remains consistent and conforms to the order in which iterations should execute. Iterations should be allowed to execute in any order but committal order should be enforced.

One way in which to enable this model of optimistic execution is to assign priorities to iterations and, while allowing lower priority iterations to completely speculatively execute, enforce that higher priority iterations always commit before lower priority. This way state consistency is ensured. Uncommitted iterations should be stored in a heap-like data-structure and only allowed to commit when at the top of the heap. This implementation nevertheless leaves the programmer with the task of ensuring that when committing the root of the heap, that iteration has the highest priority and that no other higher priority iteration will occur in the future.

Galois Implementation

Galois' optimistic execution is supported by set iteration on worklist based algorithms (chapter 4). When iterating over an ordered set, Galois' runtime system perceives this and implicitly adds the same ordering to the *commit pool*. The commit pool is the structure responsible for ensuring the order of iteration committal. The commit pool allows the runtime to speculatively execute iterations further than what would be possible by following the absolute ordering.

In practice, the priority and corresponding committal ordering is a property of the ordered set, establish via Galois's library provided ordered worklist. Priorities are introduced via a Java Comparator object with which the worklist is ordered. Implementation of the comparator is algorithm specific and the responsibility of the programmer.

As is the case of the unordered set iterator, dependences between iterations may occur and the same rollback procedures are applied. However, instead of rolling back the offending iteration, the lowest priority iteration must always rollback when confronted with a higher priority iteration. This reduces the change of deadlocking.

An example of the code needed by Galois in order to introduce order to an algorithm can be seen in **Listing 13.**

```
1 //how to compare elements to decide priorities
2 Comparator<Element> comparator = new Comparator();
3 OrderedWorklist<Element> wl =
        GaloisWorklistFactory.makeOrderedWorklist(comparator);
4 foreach(Element in wl){
5     work = //compute something
6     wl.add(work);
7 }
```

Listing 13 – Ordered in Galois' foreach iterator.

```
1  Graph graph;
2  Worker worker;          //worker thread
3  InOrderScheduler scheduler =//iterations from graph
4  MST mst; //minimum spanning tree;
5
6  while (true){
7     try{
8        Iteration it = scheduler.newIteration(worker);
9        do {
10          scheduler.nextElement(it);
11          inNode=it.getEdgeIn();
12          OutNode=it.getEdgeOut()
13          tree=//See if is valid path and create MST
14          mst.replaceSubgraph(it, tree);
15       } while(it.workLeft());
16       //commit this iteration if it is top priority.
17       //If not, commits the top of the heap.
18       scheduler.commitInOrder(it);
19
20    }catch (violationException ve)
21       //do nothing, graph is only updated on commit
22    }
23 }
```

Listing 14 – In-Order implementation of Kruskal's MST.

Related Patterns

- ***Parallel Pipes and Filters***

  With *Parallel Pipes and Filters* [90], computations are ordered but if input data is available, they remain independent from each other and can be executed in parallel. This is similar to what we propose in *In-Order Iteration.*

Known uses

On processing algorithms subject to ordering constraints, static approaches tends to provide more efficient implementations of algorithms. In cases where data dependences are only available at run-time, more careful handwritten concurrent implementations using coarse locking mechanisms are usually preferred due to the small amount of parallelism available. Therefore, the number of speculative parallelization approaches that provide support for ordering is reduced.

The SETL language [91] for set theory has tuple iterators that are somewhat similar to Galois ordered-set iterator, but contrary to Galois, the tuple set is not unbounded and SETL is not a parallel programming language. An analogous use is that of out-of-order execution, where speculative execution of processor instructions is used to reduce the amount of time for required for future instructions [92]. The Commit pool structure is a new take on Tomasulo's *reorder buffer* [68]. Another approach adds speculative parallelization to FORTRAN-style DO-loops in X10, with resource to hardware transactional memory [93]. Safe futures are a related form of allowing for speculative ordered execution [94]

7.3.2 **Graph partitioning**

Also Known As

Distributed Graph Partitioning

Problem

How to partition a graph data-structure in a way that promotes locality-aware amorphous data parallelism?

Context

This pattern implies the previous application of *Amorphous Data-Parallelism* for irregular algorithms and of *Data-Parallel Graph*.

On parallelizing graph-based algorithms, one must consider how to exploit the structure of data to uncover latent parallelism. Graphs provide a very useful abstraction when it comes to parallel processing. Recall that a graph is a complex data-structure composed of nodes connected by edges. Nodes represent data elements while edges represent relationships or dependences among nodes.

To parallelize a graph-based algorithm, programmers must first decompose the data-structure into independently executing *partitions* and distribute them to the processing units. A partition is a group of nodes that share common traits and have tighter dependences with other nodes in the partition than with nodes belonging to other partitions. Partitioning allows the problem to be broken down into smaller and less complex sub-problems that can be handled concurrently.

Efficient execution of partitioning techniques requires workload balance and minimum inter-partition communication. In addition, partitioning strategies should be configurable so as to consent different graph topologies and allow the algorithm to run on different hardware topologies.

Forces

- ***Partition size vs. independence***

  Bigger partitions reduce the chance of conflicts occurring in inner nodes but are also less likely to be independent from other partitions.

- ***Partition size***

  Smaller partitions allow for better distribution of work among the processing units.

- ***Cost of dynamic partitioning***

  The overhead of constant repartitioning in dynamic partitioning approaches might reduce the benefit of re-distributing work.

- ***Underlying data-structure***

  The data structure must handle partitioning in an efficient way and without much computational cost.

Solution

An efficient *Graph Partitioning* approach requires the programmer to take into consideration the following steps:

Step 1 - Define cardinality

The problem of partitioning a graph entails finding the group of independently executing partitions such that workload and number of nodes is equivalent for each partition.

Efficient partitioning schemes ensure load-balancing and aim at increasing intra-partition dependences while at the same time minimizing dependences between different partitions. The number of partitions is usually a function of the cardinality of processing units. As such, the actual number of partitions should never be lesser than the number of processing units.

Step 2 - Determine the type of algorithm operator

Following the characteristics described in *Categorization of irregular algorithms* (section 2.3) the operator of an algorithm can be one of three types: *Morph, Local Computation* and *Reader*.

Step 3 - Determine the type of partitioning required by the operator

We can further classify algorithm operators by the type of partitioning techniques it requires:

Dynamic partitioning

Whenever a new node or edge is inserted into the graph, the partition changes and distribution can become highly unbalanced. Therefore, as the structure of data cannot be predicted Morph algorithms require dynamic load-balancing partitioning techniques.

If the structure of the graph is updated regularly, as is the case of Delaunay Triangulation, then the programmer must choose a low cost balancing technique that takes into account the current distribution and updates it accordingly [95-97], instead of repartitioning and redistributing the graph in its entirety. This usually means using *local improvement partitioning methods* [98]. A local improvement algorithm takes a pre-formed partition and redistributes nodes so as to achieve an optimal local partition.

Static and semi-static partitioning

On local computation and reader algorithms, the partition can be determined at the beginning of execution because the structure of data will remain the same throughout the execution of the algorithm. This means that heavyweight non-dynamic partitioning methods can be applied.

Step 4 - Choose partitioning algorithm

Graph partitioning is a very well known NP complete problem that has been the focus of much attention by the programming community. As such, many different methods and algorithms have been proposed. Describing the full set of partitioning techniques is beyond the scope of this pattern and therefore we submit further analysis to the *known uses* section of the pattern.

Step 5 - Handle Amorphous Data Parallelism

If the number of partitions is greater than the number of processing units, then can place multiple partitions on each processing unit, allowing us to fine-tune load balance by implementing work-stealing techniques or multi-threading. Furthermore, having more partitions than cores decreases the likelihood of a thread running out of work.

Recall that Amorphous data-parallelism arises in situations where the irregularity of the algorithm constrains the amount of parallelism that can be achieved. In this case, partitioning allows us to overcome the unpredictability factor. Consider the example from **Fig. 21,** where (a) represents a simple graph partitioned per four processing units.

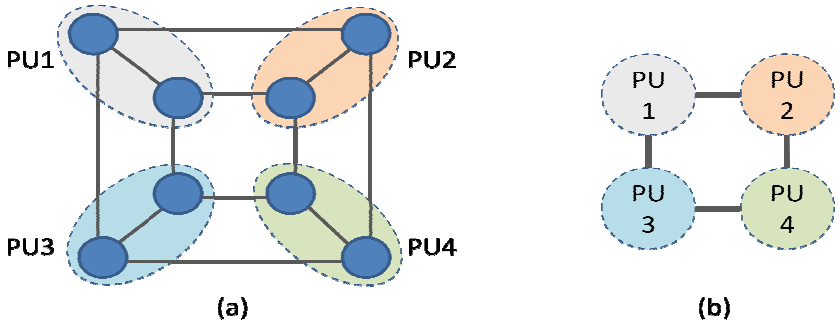


Fig. 21 – Graph Partitioning

If each partition is assigned to a single processing unit, the graph can viewed in a more abstract way as if dependences between nodes are in fact dependences between processing units (b).

Step 6 - Decide what is the execution strategy

After partitioning the algorithm, the programmer must choose how to handle algorithm execution on the partitions. This is achieved by implementing a *Graph Partition Execution Strategy.*

Galois Implementation

One of the most important ideas behind data-partitioning in Galois is that the client code should not need to change radically when going from non-partitioned to partitioned data-structures.

Galois supports partitioning at two levels: first the graph is partitioned into abstract domains. Abstract domains are then bound to the actual processing units. Each processing unit executes multiple abstract domains but each abstract domain is processed by a single processing unit. This characteristic means that the data-structure can be partitioned into more partitions than the number of available processing units and distribution of work is more evenly-balanced.

The source code required to partition a graph data-structure in the Galois framework is shown in **Listing 15 .** This code should be introduced in the initialization phase of the algorithm, when the graph class is instantiated. The programmer must provide the graph class with an instantiation of a partitioner and request it to partition the graph. Partitionable graphs implement the *Partitionable* interface. Nodes and edges in partition graphs must implement the *PartitionObject* interface, which allows the programmer to access information about the partition to which the object belongs.

```
1  PartitionedGraph graph;
2  graph.setPartitioner(new Partitioner());
3  graph.partition();
```

Listing 15 – Graph Partitioning in Galois.

As for partitioners, Galois currently supports two type of partitioning methods: *Graph bisection*, where the graph is traversed breadth-first from an arbitrary boundary node until half the nodes have been traversed, and a *Metis-based graph partitioner [99-100]*, which coarsens the graph by edge contraction and then partitions the coarsened graph.

Different partitioners can be implemented by the programmer, in order to suit specific needs of algorithms.

Example

Considering the *Preflow-Push algorithm* described in section 4.2 :

Step 1 - For the purpose of this example, let us consider a cardinality of 4, that is, the graph data-structure is to be divided in four partitions.

Step 2 - Since in *preflow-push* any changes applied to the graph occur as updates to the label of a node or to the flow of an edge, we can classify this operator as a local computation.

Step 3 - Local computation operator means that we can partition the graph a single time and the partitions will remain balanced throughout the execution of the algorithm.

Step 4 and 5 - Given the characteristics of this algorithm, we can use a partitioning algorithm that has a higher computational cost but provides an optimal distribution of nodes per processing units. Nevertheless, we could use any number of partitioning algorithms for *Preflow-Push*.

Sometimes deciding on the partitioning method is not as trivial as it seems. **Fig. 22** shows two different ways to partition a trivial *Preflow-push* graph per four processing units. In (a), although we have greater locality, since partitions are formed in the direction of the sink, the number of independent inner nodes is small. In (b) however, the number of independent nodes is significantly higher than in (a), but some of the processing units might be idle for a long time because flow has not reached the nodes in their partition. Therefore, in the case of *Preflow-Push* we have to balance locality with the amount of available parallelism.

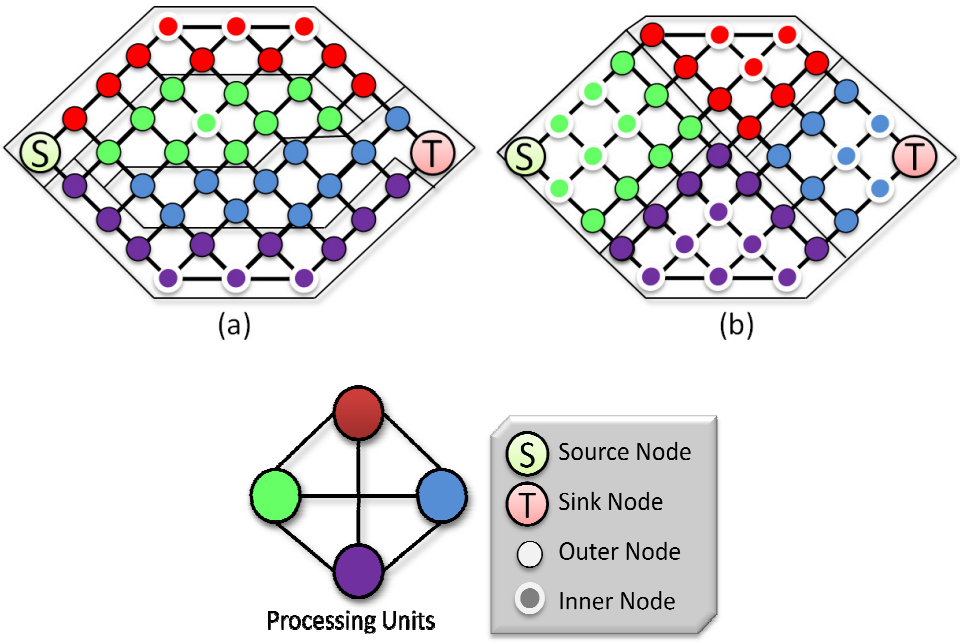


Fig. 22 – Partitioned Preflow-push graph

Related Patterns

- ***Data-Parallel Graph***

  Data-Parallel Graph provides the graph data-structure to be partitioned by *Graph Partitioning*.

- ***Geometric Decomposition***

  Geometric decomposition [56] allows programmers to decompose a data-structure into independent chunks that can be executed concurrently.

- ***Graph Partitioning Implementation Strategy***

  Graph Partitioning Implementation Strategy tackles the problem of exploiting concurrency by partitioning [67]. However, contrary to our pattern, *Graph Partitioning Implementation Strategy* is more focused on algorithms used to handle partitioning.

Known Uses

The number of algorithms available for graph partitioning is extensive but some studies of partitioning methods are well known to the parallel programming community. Karypis and Kumar [100] provide an analysis of current partitioning techniques for irregular algorithms. Wider surveys of graph partitioning algorithms are described by Fjallstrom [98] and Elsner [101].

The concept of supporting partitioning in languages and frameworks is around since the Ada language [102]. Recent approaches to high performance computing, such as High Performance Fortran (HPF) [103], Threaded Building Blocks (TBB) [104] or Chapel [105], also provide partitioning strategies. HPF focuses on the partitioning of arrays to distributed memory computers, while TBB only supports static partitioning. Chapel belongs to a group of Partitioned Global Address Space (PGAS) languages which have a partitioned memory model [106]. On these languages, a data structure is accessed as if it was local though it is in fact distributed. Chapel supports traditional data distributions as part of its class library and allows programmers to implement application specific distributions if needed.

7.3.3 Graph Partition Execution Strategy

Irregular algorithms need special executions strategies when handling a partitioned graph.

Problem

How to handle partitions while accounting for *Optimistic Iteration* of algorithms.

Context

Recall that amorphous data-parallelism often arises on worklist-based irregular algorithms whose underlying data-structures are dynamic. Executing these algorithms requires *Optimistic Iteration* techniques. These techniques assume that data dependences are not violated but provide recovery methods for the cases when they do occur.

Using *Graph partitioning*, the graph data-structure is divided into groups and distributed to the processing units. These groups of nodes are comprised of inner nodes, which are independent from nodes in other partitions, and outer nodes, which require *Optimistic Iteration* to overcome inter-partition dependences. Nodes forming a partition have thigh dependences among them.

In this context, some considerations must be made in order to ensure that *Optimistic Iteration* takes full benefit from *Graph partitioning*.

Forces

- ***Implementation cost vs. benefit***

  Implementing more efficient

- ***Lack of Optimism***

  Having an execution strategy of how to handle partitions is important even if there is a small amount of parallelism deriving from Optimistic Iteration. It's a matter or adapting the strategy to the amount of available parallelism.

Solution

Speculatively executing irregular algorithms in partitioned spaces, forces the programmer to consider how to handle different partitions. A general solution is comprised of the following steps:

Step 1 - Decide how to handle border nodes based on the type of operator

While inner partition nodes can be executed independently in parallel, bordering nodes are subject to all the constraints of *Optimistic Iteration.* Thus, the programmer must design a strategy of how the system handles these nodes.

When the set of neighbors of an iteration executing in a partition contains nodes belonging to another partition, then we have an inter-partition conflict.

Single copy – each partition has its set of bordering nodes and only that partition is able to access the nodes in the partition. In the case of an inter-partition conflict, the conflicting iteration has to request access to the nodes in the other partition. There are three main methods by which to access bordering nodes on another partition:

Migration, i.e. conflicting nodes migrate from one partition to the other, which will result in increased re-partitioning effort.

Shadow copy, where the conflicting iteration requires the locks to those specific nodes, retrieves a copy of the nodes, updates them and returns them to the original partition where they will replace the original nodes. This method involves more computational effort since each partition must know which locks are currently held by other partitions and try and execute only independent nodes.

Lock acquisition, where the conflicting iteration simply requests the lock to the entire partition and updates it at will. This method has less computational effort since it only acquires the locks a single time. However, this approach highly reduces the amount of available parallelism as an entire partition is locked when only a sub-set of its nodes is required.

All these methods incur in an additional overhead derived from having to make requests to the processing unit that holds a given partition. If that processing unit is currently executing an iteration on the partition we want to access, then the conflicting iteration has to rollback and try to execute at a later stage. Therefore, scheduling has to be partition aware.

Redundancy – bordering nodes can have redundant copies on each adjacent partition. Using redundancy for non-reader algorithms requires the programmer to implement strict synchronization mechanisms to keep copies coherent. On large scale graphs, maintaining coherence can quickly become computationally intensive. However, if we consider only

reader algorithms, using redundant copies of border nodes is a useful strategy to handle partitions. The overall strategy is depicted in **Fig. 23**.

This is a good solution if neighborhoods can only be comprised of adjacent nodes. If not, the programmer would need to increase the size of the redundantly copied neighborhood. This can quickly become a problem since redundancy has a heavy memory cost. However, with redundant copies, every node can be executed independently, which at the same time removes the need for strict scheduling.

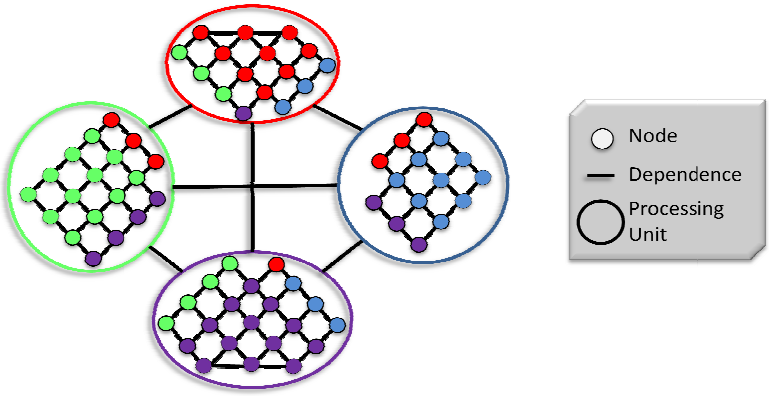


Fig. 23 – Redundant copy of bordering nodes

Step 2 - Handle graph dynamism

Irregular algorithms are characterized by dynamic changes in the structure of data. When considering partitioning on algorithms with morph operators, the programmer has to determine how to handle the various updates to the data-structure.

New data elements can be added to partitions arbitrarily, for instance, to the partition that originated the element. This method has a small computational weight but can lead to unbalanced partitioning. Alternatively, new data elements can be added to the partition that has fewer elements, maintaining an averagely balanced distribution but risking reducing locality and negatively influencing the structure of data dependences. Another option is to repartition the data-structure with each new addition, which is computationally heavy but provides an optimally balanced partitioning.

The best solution is probably a compromise of the previous options: any new element is added to the partition that created it and after a given number of new additions, the

data-structure is repartitioned to balance the workload. This solution reduces the cost of repartitioning the structure while at the same time supporting data-locality. However, delaying the repartition might require the data-structure to be partitioned entirely to ensure balance, instead of locally repartition sub-sections of the overall structure.

None of these solutions is optimal and the programmer should decide on the compromise that better suits the algorithm.

Step 3 - Consider work partitioning

An efficient partition of an algorithm is not ensured simply by partitioning data-structures. To achieve proper division of work per processing unit, programmers should reduce access to non-partitioned data-structures, which enforce synchronization bottlenecks and reduce concurrency. On worklist-based irregular parallel algorithms (see section 3.2) this bottleneck derives from accesses to the global worklist.

The solution therefore is to partition the worklist respecting the partition of the data-structure. In the overall, partitioning the worklist adds to the locality of the algorithms. When a processing unit adds work to the worklist, partitioning ensures that that same processing unit will eventually process the work it produced.

Worklists can be partitioned in two ways:

Partitioned Global worklist – The worklist remains globally accessible although work elements are partitioned, i.e. each processing unit only receives work for the partitions it currently holds. The access to the worklist itself doesn't need to ensure mutual exclusion since only the scheduler accesses it. However, the scheduler remains a concurrency bottleneck.

Partitioned Local worklist – The worklist itself is partitioned and each processing unit maintains its own local worklist. This implementation allows the processing units to increase data locality since new work produced is placed on the local worklist. However, this approach requires additional work from the scheduler who now has to schedule iterations while keeping track of multiple local worklists.

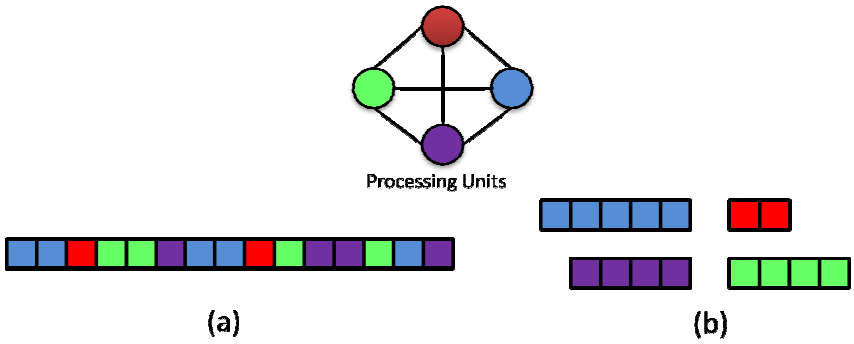


Fig. 24 – Worklist partitioning

Fig. 24 represents the two approaches described here for the data partitioning described in **Fig. 23**.

Galois Implementation

In Galois, the execution strategy that handles partition makes the following compromises:

- The runtime system's scheduler assigns work to cores dynamically, but in a partition-sensitive way. This means that the worklist remains globally accessible but that the scheduler only assigns work to the processing unit that holds the partition that the work will affect.
- If a conflict arises with nodes in other partitions, the conflicting iteration requests the lock for the entire partition and execution proceeds according to *Optimistic Iteration*. If the partition is not in use, the iteration can acquire the lock and continues to process the iteration. If the lock is already acquired, one of the iterations has to rollback. Locks are released when the iteration either completes or aborts.
- When a processing unit is executing work from a partition it currently holds, and if none of the neighborhood nodes is an outer node, both lock acquisition an conflict detection are disabled for the duration of the iteration.
- A partition's outer nodes and edges implement *BoundaryObject* interface, which provides an extra labeling property that defines if they are boundaries.
- New data elements added to the data-structure are inserted in the partition of the iteration that created them. However, the programmer can override this implementation.

The code needed to instantiate the general Galois partition execution strategy is described in **Listing 16.** This code should be introduced to the algorithm just after initializing the graph partition. The programmer must first instruct the runtime system to recognize the different partitions and avoid conflict detection on inner nodes (line 1). The partitioned worklist is created via the GaloisWorklistFactory. However, creating a partitioned worklist requires the programmer to convert all work objects to *PartitionObject* (line 6-8). Contrary to normal worklist elements, *PartitionObjects* know which processing unit currently holds the partition they belong to.

```
1  GaloisRuntime.setupPartitioning();
2  UnorderedWorklist<PartitionObject> ws =
                GaloisWorklistFactory.makePartitionedWorklist();
3  ArrayList<Node<PRFNode>> workset = // initialize worklist
4  //initialize the partitioned worklist
5  //objects must be converted to PartitionObject
6  for (Node n : workset) {
7       ws.add((PartitionObject) n);
8  }
```

Listing 16 – Galois partition execution strategy.

Example

Using the *Preflow-Push algorithm* described in section 4.2 and following the sequence of steps considered in the *Graph Partition Execution Strategy* solution we could reach the following compromises:

Step 1 - In *Preflow-Push*, the operator is a local computation. In this case, updates to nodes' labels is not computationally expensive and so the best way to handle outer nodes in this algorithm is by acquiring the lock to the entire partition. This way, and since this method provides lightweight locking, there isn't a big disparity between the cost of locking and the cost of computing the new labels.

Step 2 - As was stated above, the operator of this algorithm is a local computation and therefore we need not concern with graph dynamism.

Step 3 - On this algorithm, iterations will tend to traverse the data structure in a wave-like pattern, from the source to the sink. This means that there will probably be a great number of conflicts as iterations progress side by side. Therefore, the main

concern should be to reduce the cost of scheduling iterations and so the worklist should remain global.

Related Patterns

- ***Graph partitioning***

  Graph Partition Execution Strategy requires that a graph-like data structure be partitioned by *Graph Partitioning.*

- ***Optimistic Iteration***

  When *Optimistic Iteration* concepts are applied to partitioned graphs, programmers require valid *Graph Partition Execution Strategies.*

Known uses

A similar approach to worklist partitioning is proposed by Chandra *et al* [107]. Their dynamic partitioning strategy named Dispatch builds processor-local worklists. Each processor then reconstructs the global work distribution list from the set of local worklists. A similar approach was used by Bai *et al* [108] to develope a software transactional memory executor that partitions transactions among processors by grouping them based on their search keys.

The Chapel programming language [105] uses an asynchronous partitioned global address space (APGAS) programming model which provides virtual partitioning of data-structures in memory spaces. This is an analogous yet very different approach to partitioning. In Chapel each processor node retrieves task from a task pool but can also invoke work on other processor nodes using *On clauses.* These force computations to occur in the processor node that holds the object in memory. Processor nodes can also fetch data from remote locations.

7.4 Algorithm Optimization Patterns

7.4.1 One-Shot

Also Known As

Optimistic Conflict Detection

Problem

How to reduce overhead derived from conflict checking operations in optimistic approaches?

Context

Optimistic Iteration needs to assume that dependences are maintained throughout the execution of the application. However, to guarantee that no violations of this pre-condition occur, the system makes dependence checks at each iteration. When a violation occurs, the iteration is halted and appropriate corrective actions are taken. If no violations are detected, the iteration is allowed to commit its results. On detection of a dependence violation, the activity that was being performed is halted and all the computational effort spent in its execution is wasted. Furthermore, it incurs additional overhead from actively rolling back the computation.

Recall that the set of neighbors of an iteration is composed of every element that might be read or written by a computation, meaning that reducing the number of conflicts implies locking the entire neighborhood of a computation.

Forces

- ***Abort Ratio***

  If the baseline implementation of this algorithm has a high abort ratio, then there will be little gain in implementing the *One-Shot* pattern.

- ***Locking Overkill***

  As the neighborhood of any computation is always lesser than or equal to the entire data set, a neighborhood containing the entire data set is an inaccurate but correct approximation. Therefore, this pattern can be used for any algorithm, with various levels of effectiveness.

- ***Overzealous Neighborhood vs. Available Parallelism***

  A bigger, more cautious, estimation of the neighborhood can reduce the probability of unaccounted data-accesses but will in turn reduce the amount of available parallelism, since there are less available elements to process.

- ***Restrained Locking***

  If the neighborhood was wrongly estimated, then there will be undetected conflicts, which will in turn increase the number of aborted activities and can lead to undesirable results.

- ***Optimization Gain vs. Number of Threads***

  The amount of time needed by an iteration prior the disabling of conflict detection is relative to the number of locks and the abort ratio. Therefore, if many threads are active, there will be a larger abort ratio and the conflict detection for a given iteration will take longer to be disabled.

Solution

If the neighborhood of a computation can be statically predicted in a straightforward manner, without any additional computation, then the neighborhood can be immediately locked at the beginning of the iteration and the conflict manager can be henceforth disabled.

The following steps are strongly related to those of *Optimistic Iteration.* By applying these steps, the overhead due to aborted activities is significantly reduced:

Step 1 - Determine the type of algorithm operator

For this specific pattern to have some benefit, the operator must introduce some sort of data-dependence and therefore we're only interested in non-reader algorithms.

Step 2 - Predict the set of neighbors of each iteration

If the set of data elements that need to be locked is easily estimated, or if it is not but there is a logic over approximation that might prove equally efficient, that allows the programmer to introduce this pattern. Furthermore, the programmer must also take into account that an inaccurate or too constricted prediction of the neighborhood may not only restrict the amount of available parallelism but will in fact increase the number of aborted iterations, thus reducing performance.

Step 3 - Lock all affected neighbors

Lock the set of neighbors with the locking mechanisms provided by the implementation language or framework.

Step 4 - Disable conflict detection for Optimistic Iteration

Stop detecting conflicting data accesses and assume they are not violated.

Galois Implementation

The Galois implementation of this pattern requires the programmer to specify that the algorithm implements the *One-Shot* pattern by:

1. A client call to all neighbors of the active node, which the Galois' Runtime System will intersect and order the lock of these same neighbors to be acquired.
2. Disabling the Conflict Manager using the Singleton class ConflictManagementSwitch, which provides a single point of access to the current iteration's Conflict Manager. This is achieved by setting the disable conflict management switch to true for the current iteration.

The source code required to introduce the *One-Shot* pattern in the Galois framework is shown in **Listing 17.** This code should be introduced within the foreach loop, preferably at the beginning, since the programmer must minimize the number of computations performed before disabling the conflict manager.

The current implementation of Galois does not fully justify the call to `getOutNeighbors` without a deeper understanding of the locking and conflict detection strategies of the runtime system. In this case, this call intends to "touch" all neighbors so the runtime of Galois knows what elements should be lock for a given iteration of the algorithm.

```
1 foreach (Node in graph){
2    Collection<? Extends Node<NodeData>> neighbors
                              = graph.getOutNeighbors(node);
3    ConflictManagementSwitch.disableConflictManagement(true);
4    //do computations
5 }
```

Listing 17 – One-Shot pattern in Galois.

Example

In the *Preflow-push Algorithm*(section 4.2 the neighborhood of an activity can only consist of its incoming and outgoing edges, with the addition of its downstream neighbors. This means that, for a given active node, its neighborhood can be accurately estimated without any computation and therefore locking can be disabled at an earlier stage of the iteration.

Counter-Example

In *Delaunay Triangulation* (section 4.1 the actual neighborhood of an active node cannot be determined prior to the actual triangulation. In this case, the active node corresponds to a triangle (group of three points) and the neighborhood of an active node corresponds to all neighboring triangles of the active triangle and any triangles affected by edge flipping.

Related Patterns

- ***Iteration Chunking***

  Combined with *One-Shot*, the Iteration Chunking pattern allows for an even greater performance improvement over the baseline implementation.

- ***Optimistic Iteration***

  The *Optimistic Iteration* pattern is a pre-requisite for the implementation of the *One-Shot* pattern.

Known Uses

Since predicting the neighborhood in irregular algorithms is equivalent as using static analysis to uncover data dependences in regular algorithms, we can say that *Pessimistic Concurrency Control* [109], a model used by the transactional memory community, is analogous to *One-Shot*. Pessimistic locking involves detecting which resources are used by a transaction and locking that same resources for the exclusive use of that transaction. The resources then remain locked until the transaction either commits or rolls back. This is also analogous with the concept of *atomic sections* [110] of code, where all updates to the code in that section are viewed as occurring at the same instant in time. Cherem *et al* [111] present a Java framework for inferring atomic sections which behaves similarly to what is described for One-Shot in the Galois framework. They developed a protocol that estimates what resources are accessed in the atomic section, and introduce locks at the beginning of the said section.

For the Galois framework, *One-Shot* execution was first described by Méndez-Lojo *et al* [112].

7.4.2 Iteration Chunking

Also Known As

Iteration Coalescing, Data Loop Chunking

Problem

How to reduce the overhead derived from the dynamic assignment of work to threads? How to improve on the locality of geometrically adjacent iterations and reduce constant acquiring and releasing of lock on common elements?

Context

This pattern requires the previous implementation of *Amorphous Data-Parallelism*.

On working with concurrent worklist-based iterative algorithms (see section 3.2), there are three major sources of overhead derived from accessing the worklist in a dynamic fine-grained manner:

- Every time a thread needs to access the worklist, he needs to acquire a lock to the worklist, retrieve the element and release the lock, so that other threads are able to access it. If the worklist is already locked by another thread, the accessing thread must wait until the lock is released.
- The majority of these algorithms have low computational weight per iteration while the weight of locking the data elements is usually high. Consequently, the algorithm has a higher overhead in locking than actually performing the computation.
- As different iterations retrieve data elements from the worklist, there is little concern as to whether the subsequent iterations performed by a thread benefit from the *principle of locality* [113]. This is especially true for data dependences since if A depends on both B and C, then there is a high probability that B and C also have some degree of dependence between them. In this case, blindingly assigning iterations to threads increases the number of cache misses.

Although each algorithm incurs in its own set of overhead causes, this specific set is endemic to concurrent worklist-based iterative algorithms and should therefore be resolved in order to achieve the maximum desired performance.

Forces

- ***Workload Distribution***

  Every thread should maintain a similar workload, despite the reorganization of work assignment derived from the application of this pattern.

- ***Density of Data dependences***

  The number of actual data dependences influences the optimization. If there are too many data dependences between iterations then the amount of parallelism we are able to extract is constrained by the number of locks. If the number of data dependences is too few, then there is not much gain in chunking iterations.

- ***Cost of Computation***

  If for every iteration the cost of computation is greater than the cost of locking elements, then we must consider if there is any advantage in chunking since chunked iterations will be remain computationally heavier while the cost of locking is reduced.

While loop chunking only needs to be performed if the body of the loop is so small that the execution time is dominated by speculation overheads

Solution

While programmers are taught to introduce parallelism in fine-grained amounts, there is much to be said, performance wise, about combining multiple fine-grained blocks of work into a single coarse-grained block. This is true especially if there is a significant number of tightly data-dependent iterations whose concurrent execution would inevitably result in conflicts. There are at least two possible solutions to correct this overhead: global worklist chunking or using thread-local worklists.

Global worklist chunking

The simpler solution is to have each thread pick multiple work items from the worklist, creating a super iteration composed of multiple iterations. This solution however does

not allow us to achieve significant gains in data locality, since, if a thread retrieved N work items from the worklist, the likelihood of all N iteration chunks having sharing at least one data element is small. This means that cache misses would still occur and there would be few to none shared locks between the chunks. **Listing 18** shows how a simple dynamic iterative code (a) can be transformed by Iteration Chunking (b).

Thread-local worklist chunking

Another solution is to have each thread handle the work they themselves create. This means that this solution cannot be used for algorithms such as *Cholesky Factorization*, since no new work is added to the worklist.

As described in **Listing 19**, this solution is implemented by instantiating a thread-local worklist on which all work created by that thread is placed. The thread instead of fetching work from the global worklist and incur in concurrency problems, now fetches work directly from its local worklist. An additional advantage is that all new work that created tends to share some data elements with the work that created it and therefore data locality is increased. This fact is not true for all irregular algorithms, since the new work created might not be adjacent to the source and will therefore not benefit from this approach.

```
Worklist wl;
for each element in wl
{
    lock data elements;
    work=compute something;
    wl.add(work);
    unlock data elements;
}
```

(a)

```
Worklist wl;
for index < wl.size(); index++
{
    for index<index+5; index++ {
        element = wl.get(index);
        lock data elements;
        work = compute something;
        unlock data elements;
    }
    wl.add(allWork);
}
```

(b)

Listing 18 – Chunking of Iterations with global worklist.

```
1   Worklist wl;
2   for each element in wl
3   {
4      Worklist local;
5      local.add(element);
6      while(local.notEmpty()){
7         lock data elements;
8         work = compute(local.getNext());
9         If(work!=null)
10           local.add(work);
11        unlock data elements;
12     }
13  }
```

Listing 19 – Chunking of Iterations with thread-local worklist.

None of these solutions can be effectively used if there is a restrict iteration order. Another important consideration is that although locking a bigger set of data elements, which is in fact consist on the union of the active elements of every chunked iteration, we remove the chance of arising conflicts between chunked iterations while, at the same time, increasing the likelihood that a conflict will occur with other concurrent iterations.

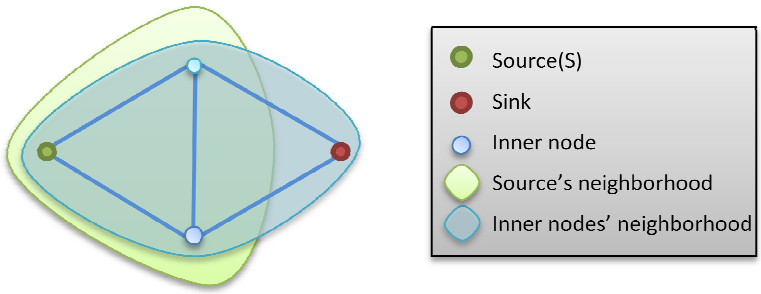
Galois Implementation

Global worklist chunking is difficult to implement in Galois since retrieving elements from the worklist is performed by the scheduler and not actively fetched by the thread. Therefore, Galois uses the *Thread-local worklist chunking* option, with some small considerations related with optimistic view of execution. The first consideration is on how Galois handles the increase of conflicts due to the increase of neighborhood size. In this case, Galois must ensure that on an iteration composed of chunked iterations, if a conflict arises in the execution of any of the chunks, only that execution is rolled back. All previously completed iterations must be committed since they completed correctly. The act of committing the iterations is completely handled by the runtime. The programmer must only take care to, on detecting a conflict, place all remaining local work on the global worklist. Additionally on Galois, the number of iterations that can be chunked at one time should be pre-set, so as to prevent and iteration from getting too much work and locking too many neighborhoods. In this case, when the number of chunked iterations on a thread reaches the pre-set maximum *chunking factor*, all work left on the local worklist is placed in the global worklist and optimistic iteration is

resumed. Furthermore, the Galois implementation of this pattern requires *Lock Reallocation* as a way of reducing the amount of locking operations.

Example

The *Preflow-push Algorithm* (section 4.2 is a good candidate for this optimization since when new work is created, there is a good chance that the data elements they need will overlap. As an example, consider the case of a simple graph containing only the source and sink nodes, plus two other inner nodes.



In this example, if a thread processes the source node and then adds both inner nodes to its local worklist, as there is a great deal of overlap in the neighborhoods and both inner nodes have the same neighborhood, the thread needs only acquire one extra lock when handling the inner nodes. At the same time, as all elements are already in cache, this adds a great deal of data locality.

On the *Delaunay Triangulation Algorithm* (section 4.1 since the algorithm already uses a local worklist on the inner loop, *Iteration Chunking* could only be implemented using *Global worklist chunking.*

Counter-example

On the *Sparse Cholesky Factorization Algorithm* (section 4.3 , this optimization cannot be accomplished due to the ordering imposed on the iterations, which would remove any chance of concurrent execution if chunking where to occur. There are some algorithms that when an iteration creates more work that usually happens in a non geometrically adjacent spaces. In *Ray Tracing algorithms* [114], for instance, since the objective is to trace the path of light through a scene, new work is usually created far from the light source.

Related Patterns

- ***Optimistic Iteration***

  The *Optimistic Iteration* pattern is a pre-requisite for the implementation of the *Iteration Chunking* pattern.

- ***Lock Reallocation***

  If applied together with *Iteration Chunking*, there is no need to release locks before committing the iterations. The use of *Lock Reallocation* is enforced in the Galois implementation of *Iteration Chunking***.**

Known uses

The idea behind *Iteration Chunking* is abundant in task parallel languages is analogous to that of *Task Chunking* [115-116] and *Loop Chunking* [117-118]. Task chunking considers how to improve on task parallelism by joining multiple tasks into one master task. This is akin to the approach using the worklist to distribute work to the threads. Loop chunking is more related to iterative processing and on how to join multiple loops, to increase performance. There are also many compiler directed scheduling strategies for the improvement of data locality in tasks using chunking techniques [119-120], and for *Thread level speculation* [76, 121-122], whose execution model is similar to that of Galois.

7.4.3 **Preemptive Read**

Also Known As

Cautious Iteration

Problem

How to reduce overhead derived from rollback operations in optimistic approaches?

Context

Optimistic Iteration assumes that dependences are not violated in the course of the execution of the algorithm. Still, locking mechanisms must be applied in order to prevent illegal concurrent access to the data elements currently being processed. If a thread tries to access an element whose lock is held by another thread, a violation occurs and the iteration must

rollback. Rollback operations undo any modification performed by both local computation and morph operators (section 2.2) by applying the inverse method of that operator.

How then can we reduce the overhead of rollback operations on concurrently executing iterations.

Forces

- ***Abort Ratio***

  If the baseline implementation of this algorithm has a low abort ratio there is little gain in implementing *Preemptive Read*.

- ***Restrained Locking***

  If the set of neighbors of an iteration is wrongly estimated, applying *Preemptive Read* will cause undesirable results due to unchecked conflicts.

Solution

Consider the following iteration examples:

```
1  a=dataStructure.get(0);
2  dataStructure.set(0,c);
3
4  b=dataStructure.get(1);
5  dataStructure.set(1,a);
```

(a)

```
1  a=dataStructure.get(0);
2  b=dataStructure.get(1);
3
4  dataStructure.set(0,b);
5  dataStructure.set(1,a);
```

(b)

In iteration (a), reads are interleaved with writes. If a conflict arises on the second access to the data-structure (line 4), causing this iteration to rollback, then resetting the iteration to its initial state means executing the inverse method `dataStructure.set(0,a)`. However, if the programmer places all reads on the beginning of the iteration, all conflicts that are able to cause a rollback operation occur on that read-only phase and, since no update was ever made to the data-structure, there is no need to perform a rollback. The solution therefore is to move all read operations to the beginning of iterations.

However, if iterations are *In-order*, this solution should not be applied. Consider the case were the iteration that is conflicted is already on the write phase but has a lower priority. In

this case, whether the iteration has a read phase or not does not influence the outcome and the iteration will rollback.

Galois implementation

In Galois, using preemptive reads requires the programmer to specify the execution mode of the runtime system as *"cautious"*. When defining the properties file for each implementation, the programmer decides whether the execution mode is cautious or standard. On a cautious implementation, the conflict manager does not store undo information for the iterations, assuming that all conflicts will arise on the read-phase and that if the conflict occurs at a later point, the conflicting iteration has to rollback. This also means that after the read-phase, when the runtime system detects the first update to the data-structure, it disables the conflict manager for the rest of the iteration.

Preemptive Read is especially effective in Galois implementations because locking is implicit. The runtime captures accesses to data elements and triggers the lock, whether that access is a read or write operation.

Example

The majority of irregular algorithm implementations traditionally perform *preemptive reads.* The algorithms described in chapter *4* are clear examples of *Preemptive Read.*

Consider the explicit locking implementation *Preflow-Push* iteration in **Listing 20**. Although the first operation relabels the node (line 5), that node is already locked because it is the element retrieved from the worklist (line 3). Next, a read operation (line 6) creates a list of neighbors that will be iterated by the loop. The instantiation of list with every node that might be updated by the iteration allows us to lock the entire set. Since every subsequent write operation affects only elements already belonging to the neighbors list, we ensure that this iteration will never try to access a data element whose lock is held by another iteration.

```
1  Worklist wl = new Worklist( graph) //create worklist
2  for each Node node in wl do{
3     lock(node); //lock node
4     //try to relabel the node
5     graph.relabel(node);
6     List neighbors = graph.getNeighbors( node);
7     lock(neighbors);//lock all neighbors
8     //try to push flow to every neighbor
9     for( Neighbor ng in neighbors){
10        //from this point on conflict manager if off
11        if( graph.canPushFlow(node, ng)){
12            graph.pushFlow(node, ng);
13            if ( ! ng.isSourceOrSink())
14                wl.add( ng);
15            if ( ! node.hasExcess())
16                break;
17        }
18    }
19    if ( node.hasExcess())
20       wl.add( node);
21    unlockAll();//release all locks
22 }
```

Listing 20 – PreflowPush algorithm with explicit locks.

This is an explicit lock implementation equivalent to the implementation in section 4.2 . Contrary to the implicit locking mechanisms provided by Galois' runtime system (section 0), explicit locking allows us to consider when and where locking mechanisms affect the code.

Related Patterns

- ***One-Shot***

  One-shot similarly disables conflict detection, although this is explicitly done by the programmer.

- ***In-order Iteration***

  In *In-order Iteration* conflict detection cannot be disabled because the iteration might rollback if a conflict occurs with a higher priority iteration.

Known Uses

Preemptive Read or *operator cautiousness* was first identified by Méndez-Lojo *et al* [112] as a general property of irregular algorithms. However, preemptive reads have been regularly used by the parallel programming community to implement irregular algorithms [4, 37, 60, 123].

7.4.4 Lock Reallocation

Also Known As

Computation Lock Coarsening

Problem

How to reduce the number of locking operations per iteration and reap the benefits of lock-oriented temporal locality?

Context

On concurrent worklist-based iterative algorithms, each iteration must acquire locks for the data elements it requires, perform some computation and then release the locks. This sequence of steps is the same whether subsequent iterations on a thread need to lock the same data elements or not.

Forces

- ***Cost of Overlap Checking***

  If for any two iterations executed in sequence on a single thread, the set of elements each requires has little or no overlap, then the computational cost of checking for overlaps might be more than the cost of releasing and reacquiring the same locks.

- ***Overextending exclusion***

  If one of the locked data elements is seldom or never accessed by an iteration or if it is only used for reading, then holding that lock for an extended time will not actively progress the algorithm. Furthermore, holding the lock will prevent other threads from accessing the object and therefore reduce the amount of available parallelism.

Solution

Each thread needs to keep a record of every data element for which he currently holds the lock. Whenever starting a new iteration, the thread will intersect the set of locks it has with the set of locks he needs to acquire. This way, excessive locks hold by the threads are released while those for new data elements are acquired. Locks for the common elements are never released and therefore the number of acquired locks using *Lock Reallocation* is always less than or equal to the base implementation.

Handling data access conflicts is dependent on the actual implementation language or framework and as such will not be covered here.

Galois Implementation

In Galois, this pattern's application is implied by Iteration Chunking.

Furthermore, on Galois implementations where *Lock Coarsening* is applied, instead of a thread releasing the locks it holds in the end of each iteration chunk, he simply retrieves the next chunk and acquires the new locks it needs. When the coalesced iterations are ready to commit, all of the locks are released at the same time.

Example

In the example of the *Preflow-push Algorithm* (section 4.2 there is some gain in performing *Lock Reallocation,* since iterations always progress via directly connected nodes. There is great probability that succeeding retrievals of work from the worklist will have some of the same neighboring nodes.

Related Patterns

- ***Iteration Chunking***

  If applied together with Iteration Chunking, there is no need to release locks before committing the iterations.

Known uses

Diniz and Rinard [124] first introduced Computation Lock Coarsening as a way of reducing locking overheads. They developed an analysis algorithm to identify code that performed constant relocking on data elements and introduce *Lock Reallocation.* Another concept tightly

connected with this pattern is that of *Reentrant Locking*, which allows a thread to acquire a lock it already owns without having to release it first. Using reentrant locks, there is no need to keep a list of which locks a thread currently holds.

The Galois based *Preemptive Read* was first described by Méndez-Lojo *et al* [112] as a property called *lock locality* that could be used as further optimization of *Iteration Chunking*.

(Page intentionally left blank)

8 Related Work

Most patterns for parallel programming described in the literature are represented in pattern catalogues with independent or semi-independent patterns. Furthermore, they represent design patterns for parallel programming and are too tightly linked to concepts inherently related with the programming language and paradigm. Gang of Four design patterns, for instance, provide solutions that require the programmer to be aware of object-oriented concepts such as Objects and Classes and, although adaptable, each pattern presents classes, methods, and hierarchy that the programmer should follow in order to achieve the solution [6]. This fact derives from the architectural patterns, which presented complete architectural solutions that could not be independent from the materials used in the construction. Contrary to that view, our patterns represent solutions that do not state how class structure is achieved or how an object is instantiated. Our pattern language present advices and considerations and discusses how a programmer should introduce the solution and why. Our patterns are independent from how the programmer decides to implement them.

Schmidt et al [125] present a set of patterns for concurrency and networking that does not focus on semantics and domain-dependent concepts and does in fact represent a pattern language. However, as they themselves discuss, each pattern is self-contained and independently described. Therefore we do not consider this as a full fledge pattern language, but instead a pattern catalogue with inter-pattern dependences. They describe a total of eighteen patterns divided in four different concerns: Service Access and Configuration, Event Handling, Concurrency, Synchronization. An addition remark goes to the fact that as we have used Galois to instrument our patterns, so has Schmidt et al used the JAWS web server.

The pattern language proposed here has close relations to some of the pattern languages for parallel processing proposed by the software pattern community – such is the case of pattern repository of the Hillside group [67] and the pattern language of Mattson *et al* [56]. However, our view is that most pattern languages and catalogs mostly represent solutions for regular problems and handle irregularity as special cases, in which case the solution needs to conform

to a different set of characteristics. Our pattern language contrasts with this view and is specifically focused on irregular problems, which are considerably more complex. In this dissertation, we propose instead to classify the solution to regular problems as a subset of the solution of irregular problems. There are nonetheless some pattern languages designed for specific irregular algorithms, as is the case of Dig *et al* pattern language for N-Body methods [126]. N-Body is a well known irregular problem.

Mattson *et al* [56] describe a set of nineteen patterns for parallel programming. This is probably the first complete pattern language for parallel programming and illustrates some of the most well known concepts of parallelism and distributed computing as patterns. The language itself is divided in four design spaces: Finding concurrency, Algorithm Structure, Supporting Structures and Implementation Mechanisms. The Finding Concurrency design space instructs the programmer on how to structure code to expose latent concurrency, while the Algorithm Structure design space considers how the algorithm can use that concurrency. The Supporting Structures design space concerns what and how algorithm structures can be used to achieved concurrent behavior. The final design space concerns low level Implementation Mechanisms such as parallel programming languages and libraries. These design spaces are similar to our own, since they too have a tight relation with the algorithm implementation. However, the patterns described by Mattson *et al* have a more general focus than ours. In fact, our patterns can be said to be refabrications with a tighter, more specific context.

The pattern language of the Hillside group [67] is a in-development language that intends on describing parallel software development from architecture to the actual software implementation. Patterns from this pattern language are divided into five hierarchical layers: *Structural patterns* describe the overall structure of a parallel program. *Computational patterns* describe the different classes of computations and propose how they should be composed with structural patterns. *Algorithm strategy patterns* define strategies for exploiting parallelism and concurrency. *Implementation Strategy patterns* present implementation strategies for the patterns described in the level above. Finally, *Parallel Execution patterns* present basic strategies with a lower level of abstraction closer to language and hardware implementation. As before, our patterns are closely related to some of the patterns described in this patterns language, although in a more complementary way. The patterns in this pattern

language are not as coarse as ours and therefore, we cannot say they build upon one another but instead that they coexist in slightly different contexts.

Additional details about pattern similarities are described in the Related Patterns section of each pattern.

Aside from patterns, there are other approaches that describe higher level strategies for irregular algorithms: Fonlupt *et al* [127] describes a set of load balancing redistribution strategies, illustrating several algorithm formulations. Biswas et al [3] describe computing strategies in relation to specific hardware architecture. Rünger and Schwind [128] describe parallelization strategies for algorithms that contain both regular and irregular characteristics. Ansejo et al [18] present general use optimization strategies. These strategies are not as high-level as patterns but present pattern mining opportunities for future work. Additional descriptions of related work in the field of irregular algorithms and parallelization strategies in general is described in the Known Uses section of each pattern.

(Page intentionally left blank)

9 Conclusions and Future Work

This dissertation describes a pattern language for the parallelization of irregular algorithms. In this pattern language, we have described a set of ten patterns for parallelizing irregular algorithms. The patterns are divided in three design spaces according to a hierarchy of application:

- *Structural Patterns* consider how structure an irregular algorithms to take advantage of latent data-parallelism.
- *Execution patterns* describe how to guide the algorithm to explore the maximum amount of available parallelism.
- *Optimization patterns* present available optimizations for irregular parallel algorithms.

Currently, none of the three design spaces (Structure, Execution and Optimization) present a high degree of maturity considering that they don't express the full set of solutions for the parallelization of irregular algorithms. However, when considering optimistic approaches, these patterns represent a well developed and mature body of knowledge. This pattern language is part of a work in progress, a mere step towards a more developed and mature domain of pattern-oriented software development.

We make the point to note that our pattern mining methodology, using the Galois framework as case study, has proven to be effective and, as long as patterns are conceptually visible in the framework, it provides a good alternative to traditional methods, which relied on the author being a recognized authority in the domain. However, since Galois build upon the work produced by years of successful experimentation on the field of parallelization of irregular algorithms, we can confidently state that all patterns were developed in mind of the insights and experience of years of parallel software development. With this pattern language we have extracted a conceptual framework from a concrete implementation. Moreover, while this pattern language intends to serve the community at large, it presents a valid description of the Galois framework and can potentially be used for documenting the inner detail of Galois' execution model.

9.1 Future Work

Future work opportunities concerning the set of patterns described in this dissertation are ample:

- **Mature the pattern language**

We plan to further refine this pattern language, as well as to produce a set of case studies to validate our approach. The majority of the patterns shown here were developed using solely the Galois Framework as case study [34]. Other frameworks and languages have considerably different methodologies for handling irregularity. We hope to explore these alternatives as well, and relate them to the patterns described here, enriching and maturing the language, as well as enhancing its potential applicability to cover a broader set of techniques and methods targeting parallel irregular algorithms. An additional step is to introduce a greater number of examples in each pattern, so as to better represent and validate the solution.

- **Improve the definition of Algorithmic Irregularity**

There is no consensus on the definition of irregular algorithm, though algorithm irregularity is frequently considered in the literature. From preliminary work described in this dissertation, we plan on further redefine the concept of irregular algorithm and help introduce a new definition that also encompasses non object-oriented approaches.

- **Produce implementations of these patterns using alternative approaches**

We believe that these patterns could benefit from approaches that increase the separation of concerns and concerns and allow these patterns to be applied regardless of the underlying programming language. Good approaches to this end include:

Code generation - A wizard-like tool would generate code automatically, based on user input [129]. The code would need to be matured by the programmer but the essential information would already be available.

Source to Source Transformation – Using annotations or other similar method, the programmer would mark code which would be processed by a parser and converted the code introduced by the programmer to parallel code with semantics similar to our patterns [130]. This also presents a good opportunity for automatic parallelization of algorithms.

Aspect-Oriented Programming – a rather recent programming paradigm whose objective is to provide appropriate isolation, composition and reuse of code by clearly modularizing crosscutting concerns using a new kind of module called *aspect* [131].

Skeletons – template-based method for parallel and distributed computing which hides the complexity of the underlying parallelism code [132].

- **Add Independent Development behavior to Galois**

Galois is in constant refinement and development and the downside of higher coupling of concerns is that when the underlying framework changes so does the interface with the programmer. Future Galois implementations would benefit from a decoupling of interface and framework.

(Page intentionally left blank)

10 Bibliography

[1] J. Gustafson, “Reevaluating Amdahl's Law,” *Communications of the ACM,* vol. 31, no. 5, 1988.

[2] M. Kulkarni, K. Pingali, B. Walter *et al.*, "Optimistic parallelism requires abstractions." p. 222.

[3] R. Biswas, L. Oliker, and H. Shan, “Parallel computing strategies for irregular algorithms,” *Annual Review of Scalable Computing*, 2003.

[4] M. Kulkarni, M. Burtscher, K. Pingali *et al.*, "Lonestar: A suite of parallel irregular programs." pp. 65-76.

[5] D. Schmidt, and F. Buschmann, "Patterns, frameworks, and middleware: their synergistic relationships." pp. 694-704.

[6] E. Gamma, R. Helm, R. Johnson *et al.*, “Design Patterns: Elements of Reusable Object-Oriented,” 1995.

[7] J. Noble, and A. Sydney, “Towards a pattern language for object oriented design,” *Proc. of Technology of Object-Oriented Languages and Systems (TOOLS Pacific),* vol. 28, pp. 2-13.

[8] U. Zdun, “Pattern language for the design of aspect languages and aspect composition frameworks,” *IEE Proceedings-Software,* vol. 151, no. 2, pp. 67-84, 2004.

[9] D. Roberts, and R. Johnson, “Evolving frameworks: A pattern language for developing object-oriented frameworks,” *Pattern Languages of Program Design,* vol. 3, pp. 471–486, 1998.

[10] K. Wolf, and C. Liu, “New clients with old servers: A pattern language for client/server frameworks,” *Pattern Languages of Program Design*, pp. 51–64, 1995.

[11] F. Buschmann, R. Meunier, H. Rohnert *et al.*, "A system of patterns: Pattern-oriented software architecture," Wiley New York, 1996.

[12] P. Avgeriou, and U. Zdun, "Architectural patterns revisited–a pattern language." pp. 1-39.

[13] D. Spinellis, and K. Raptis, “Component mining: A process and its pattern language,” *Information and Software Technology,* vol. 42, no. 9, pp. 609-617, 2000.

[14] P. Avgeriou, A. Papasalouros, S. Retalis *et al.*, “Towards a pattern language for learning management systems,” *Educational Technology & Society,* vol. 6, no. 2, pp. 11-24, 2003.

[15] P. Goodyear, P. Avgeriou, R. Baggetun *et al.*, "Towards a pattern language for networked learning." pp. 449-455.

[16] G. Meszaros, and J. Doble, "Metapatterns: A pattern language for pattern writing."

[17] J. Coplien, and B. Woolf, “A pattern language for writers' workshops,” *C PLUS PLUS REPORT,* vol. 9, pp. 51-60, 1997.

[18] R. Asenjo, F. Corbera, E. Gutiérrez *et al.*, "Optimization techniques for irregular and pointer-based programs." pp. 2-13.

[19] R. Das, M. Uysal, J. Saltz *et al.*, “Communication optimizations for irregular scientific computations on distributed memory architectures,” *Journal of Parallel and Distributed Computing,* vol. 22, no. 3, pp. 462-478, 1994.

[20] K. Pingali, M. Kulkarni, D. Nguyen *et al.*, “Amorphous Data-parallelism in Irregular Algorithms.”

[21] Z. Zhang, and J. Torrellas, “Speeding up irregular applications in shared-memory multiprocessors: Memory binding and group prefetching,” *ACM SIGARCH Computer Architecture News,* vol. 23, no. 2, pp. 188-199, 1995.

[22] M. Hermenegildo, “Parallelizing irregular and pointer-based computations automatically: Perspectives from logic and constraint programming,” *Parallel Computing,* vol. 26, no. 13-14, pp. 1685-1708, 2000.

[23] S. Taylor, J. Watts, M. Rieffel *et al.*, “The concurrent graph: basic technology for irregular problems,” *IEEE Parallel and Distributed Technology*, pp. 15-25, 1996.

[24] D. Nikolopoulos, C. Polychronopoulos, and E. Ayguadé, "Scaling irregular parallel codes with minimal programming effort." p. 16.

[25] L. Capretz, “A brief history of the object-oriented approach,” *ACM SIGSOFT Software Engineering Notes,* vol. 28, no. 2, 2003.

[26] E. Gutierrez, R. Asenjo, O. Plata *et al.*, “Automatic parallelization of irregular applications,” *Parallel Computing,* vol. 26, no. 13-14, pp. 1709-1738, 2000.

[27] W. Hillis, and G. Steele Jr, “Data parallel algorithms,” *Communications of the ACM,* vol. 29, no. 12, pp. 1183, 1986.

[28] J. Sack, and J. Urrutia, *Handbook of computational geometry*: North-Holland, 2000.

[29] L. J. Guibas, D. E. Knuth, and M. Sharir, "Randomized incremental construction of Delaunay and Voronoi diagrams," in Proceedings of the seventeenth international colloquium on Automata, languages and programming, Warwick University, England, 1990, pp. 414-431.

[30] J. Shewchuk, "Delaunay refinement algorithms for triangular mesh generation," *Computational Geometry: Theory and Applications,* vol. 22, no. 1-3, pp. 21-74, 2002.

[31] H. Lin, Z. Zhao, H. Li *et al.*, "A novel graph reduction algorithm to identify structural conflicts." pp. 289-289.

[32] H. Bodlaender, and B. van Antwerpen-de Fluiter, "Reduction algorithms for graphs of small treewidth," *Information and Computation,* vol. 167, no. 2, pp. 86-119, 2001.

[33] A. V. Goldberg, and R. E. Tarjan, "A new approach to the maximum flow problem," in Proceedings of the eighteenth annual ACM symposium on Theory of computing, Berkeley, California, United States, 1986, pp. 136-146.

[34] M. Kulkarni, "The Galois System: Optimistic Parallelization of Irregular Programs," Cornell University, 2008.

[35] C. Pancake, and D. Bergmark, "Do parallel languages respond to the needs of scientific programmers?," *Computer,* vol. 23, no. 12, pp. 13-23, 1990.

[36] M. Herlihy, and J. Moss, "Transactional memory: Architectural support for lock-free data structures." p. 300.

[37] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to algorithms*, Cambridge,Mass. ; London: MIT Press, 1990.

[38] G. Golub, and C. Van Loan, *Matrix computations*: Johns Hopkins Univ Pr, 1996.

[39] R. Torkildsen, "Two algorithms for the numerical factorization of large symmetric positive definite matrices for a hypercube multi-processor."

[40] M. Weiss, *Data structures and problem solving using Java*: Addison-Wesley, 1998.

[41] M. Robillard, and G. Murphy, "Representing concerns in source code," *ACM Transactions on Software Engineering and Methodology (TOSEM),* vol. 16, no. 1, pp. 3, 2007.

[42] M. Robillard, and G. Murphy, "FEAT: a tool for locating, describing, and analyzing concerns in source code." pp. 822-823.

[43] M. Robillard, and F. Weigand-Warr, "ConcernMapper: simple view-based separation of scattered concerns." p. 69.

[44] T. Schafer, M. Eichberg, and M. Mezini, "Towards Exploring Cross-Cutting Concerns."

[45] D. Janzen, and K. De Volder, "Navigating and querying code without getting lost." pp. 178-187.

[46] H. Eichelberger, *Jtransform-a java source code transformation framework*, TR 303, Institute for Computer Science, Wurzburg University, 2002.

[47] T. Reinikainen, I. Hammouda, J. Laiho *et al.*, "Software Comprehension through Concern-based Queries." pp. 265-270.

[48] C. Alexander, S. Ishikawa, and M. Silverstein, *A pattern language: towns, buildings, construction*: Oxford University Press, USA, 1977.

[49] W. Cunningham, and K. Beck, "Using pattern languages for object-oriented programs."

[50] J. Hu, “Design of a distributed architecture for enriching media experience in home theaters,” University Microfilms International, P. O. Box 1764, Ann Arbor, MI, 48106, USA, 2006.

[51] S. Henninger, and V. Corrêa, "Software pattern communities: Current practices and challenges."

[52] ACM-SIGWEB. "Hypermedia Design Pattern Repository," February, 2010; http://www.designpattern.lu.unisi.ch/PatternsRepository.htm.

[53] "Human-Computer Interaction and User Interface Design Pattern Repository," February, 2010; http://www.hcipatterns.org.

[54] "EuroPLoP Focus Group on Pattern Repositories," February, 2010; http://www.patternforge.net/wiki/index.php?title=EuroPLoP_2007_Focus_Group.

[55] F. Buschmann, K. Henney, and D. Schmidt, *Pattern-Oriented Software Architecture: On Patterns and Pattern Languages*: John Wiley and Sons, 2007.

[56] T. Mattson, B. Sanders, and B. Massingill, *Patterns for parallel programming*: Addison-Wesley Professional, 2004.

[57] M. Monteiro, and A. Aguiar, “Patterns for Refactoring to Aspects: An Incipient Pattern Language.”

[58] B. Zamani, “ON VERIFYING THE USE OF A PATTERN LANGUAGE IN MODEL DRIVEN DESIGN,” Concordia University, 2009.

[59] G. Booch, “Handbook of software architecture,” 2009.

[60] R. Lublinerman, S. Chaudhuri, and P. Cerny, "Parallel programming with object assemblies." pp. 61-80.

[61] C. Carothers, K. Perumalla, and R. Fujimoto, "The effect of state-saving in optimistic simulation on a cache-coherent non-uniform memory access architecture." pp. 1624-1633.

[62] W. Dieter, and J. Lumpp, "A user-level checkpointing library for POSIX threads programs." pp. 224-227.

[63] C. Antonopoulos, X. Ding, A. Chernikov *et al.*, "Multigrain parallel delaunay mesh generation: challenges and opportunities for multithreaded architectures." p. 376.

[64] N. Chrisochoides, C. Lee, and B. Lowekamp, "Mesh generation and optimistic computation on the grid."

[65] I. Kolingerová, and J. Kohout, “Optimistic parallel Delaunay triangulation,” *The Visual Computer,* vol. 18, no. 8, pp. 511-529, 2002.

[66] C. Verma, “Multithreaded Delaunay Triangulation.”

[67] K. Keutzer, and T. Mattson, “Our Pattern Language (OPL): A Design Pattern Language for Engineering (Parallel) Software.”

[68] R. Tomasulo, “An efficient algorithm for exploiting multiple arithmetic units,” *IBM Journal of research and Development,* vol. 11, no. 1, pp. 25-33, 1967.

[69] J. Fisher, "Very long instruction word architectures and the ELI-512." pp. 140-150.

[70] D. Jefferson, “Virtual time,” *ACM Transactions on Programming Languages and Systems (TOPLAS),* vol. 7, no. 3, pp. 425, 1985.

[71] L. Rauchwerger, and D. Padua, "The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization." pp. 218-232.

[72] M. Gupta, and R. Nim, "Techniques for speculative run-time parallelization of loops." p. 12.

[73] P. Marcuello, A. González, and D. de Computadors, "A quantitative assessment of thread-level speculation techniques." pp. 595–604.

[74] J. Steffan, C. Colohan, A. Zhai *et al.*, "A scalable approach to thread-level speculation," *ACM SIGARCH Computer Architecture News,* vol. 28, no. 2, pp. 1-12, 2000.

[75] S. Wang, X. Dai, K. Yellajyosula *et al.*, "Loop selection for thread-level speculation," *Lecture Notes in Computer Science,* vol. 4339, pp. 289, 2006.

[76] M. Prabhu, and K. Olukotun, "Using thread-level speculation to simplify manual parallelization." p. 12.

[77] Y. Zhang, L. Rauchwerger, and J. Torrellas, "Hardware for speculative run-time parallelization in distributed shared-memory multiprocessors." p. 162.

[78] L. Codrescu, D. Wills, and J. Meindl, "Architecture of the Atlas chip-multiprocessor: Dynamically parallelizing irregular applications," *IEEE Transactions on Computers,* vol. 50, no. 1, pp. 67-82, 2001.

[79] J. Oplinger, D. Heine, S. Liao *et al.*, "Software and hardware for exploiting speculative parallelism with a multiprocessor," *Computer Systems Laboratory Technical Report CSL-TR-97-715, Stanford University*, 1997.

[80] M. Dorojevets, and V. Oklobdzija, "Multithreaded decoupled architecture," *International Journal of High Speed Computing,* vol. 7, no. 3, pp. 465, 1995.

[81] J. Tsai, and P. Yew, "The superthreaded architecture: Thread pipelining with run-time data dependence checking and control speculation," *Urbana,* vol. 51, pp. 61801-1351.

[82] P. Marcuello, and A. González, "Control and data dependence speculation in multithreaded processors." pp. 98-102.

[83] J. Gross, and J. Yellen, *Graph theory and its applications*: CRC press, 2006.

[84] J. Launchbury, "Graph algorithms with a functional flavour," *Lecture Notes in Computer Science,* vol. 925, pp. 308, 1995.

[85] E. Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik,* vol. 1, no. 1, pp. 269-271, 1959.

[86] R. Pal, "RKPianGraphSort: a graph based sorting algorithm," *Ubiquity,* vol. 2007, no. October, pp. 16, 2007.

[87] S. Akl, *Parallel sorting algorithms*: Academic Press, Inc. Orlando, FL, USA, 1990.

[88] S. Das, "Adaptive protocols for parallel discrete event simulation." pp. 186-193.

[89] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” 1978.

[90] J. Ortega-Arjona, and U. Facultad de Ciencias, “The Parallel Pipes and Filters Pattern.”

[91] J. Schwartz, R. Dewar, E. Schonberg *et al.*, *Programming with sets; an introduction to SETL*: Springer-Verlag New York, Inc. New York, NY, USA, 1986.

[92] J. Hennessy, D. Patterson, D. Goldberg *et al.*, *Computer architecture: a quantitative approach*: Morgan Kaufmann, 2003.

[93] C. von Praun, L. Ceze, and C. Ca caval, "Implicit parallelism with ordered transactions." p. 89.

[94] A. Navabi, X. Zhang, and S. Jagannathan, "Quasi-static scheduling for safe futures." pp. 23-32.

[95] C. Walshaw, M. Cross, and M. Everett, *Dynamic mesh partitioning: A unified optimisation and load-balancing algorithm*: Citeseer, 1995.

[96] W. Mitchell, "A Comparison of Three Fast Repartition Methods for Adaptive Grids."

[97] G. Karypis, and V. Kumar, “METIS: Unstructured graph partitioning and sparse matrix ordering system,” *The University of Minnesota,* vol. 2.

[98] P. Fjallstrom, “Algorithms for graph partitioning: A survey,” *Computer and Information Science,* vol. 3, no. 10, 1998.

[99] G. Karypis, and V. Kumar, “METIS: Unstructured graph partitioning and sparse matrix ordering system,” *The University of Minnesota,* vol. 2, 1995.

[100] G. Karypis, and V. Kumar, “A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs,” *SIAM J. Sci. Comput.,* vol. 20, no. 1, pp. 359-392, 1998.

[101] U. Elsner, “Graph partitioning-a survey,” *time: 12/1997 Please send comments concerning this metadata document to wwwadm@ mathematik. tu-chemnitz. de last update: 9. Oktober 1998*, 1997.

[102] R. Jha, J. Kamrad, and D. Cornhill, “Ada program partitioning language: A notation for distributing Ada programs,” *IEEE Transactions on Software Engineering,* vol. 15, no. 3, pp. 271-280, 1989.

[103] C. Koelbel, *The high performance Fortran handbook*: The MIT press, 1993.

[104] J. Reinders, "Intel Threaded Building Blocks," O’Reilly Press, 2007.

[105] R. Diaconescu, and H. Zima, “An approach to data distributions in Chapel,” *International Journal of High Performance Computing Applications,* vol. 21, no. 3, pp. 313, 2007.

[106] S. Model, “Programming in the Partitioned Global Address Space Model.”

[107] S. Chandra, M. Parashar, and J. Ray, "Dynamic structured partitioning for parallel scientific applications with pointwise varying workloads."

[108] T. Bai, X. Shen, C. Zhang *et al.*, "A key-based adaptive transactional memory executor." pp. 1-8.

[109] D. Menasce, “Optimistic Versus Pessimistic Concurrency Control Mechanisms in Database Management Systems,” *INFO. SYS.,* vol. 7, no. 1, pp. 13-27, 1982.

[110] N. Lynch, W. Weihl, and A. Fekete, *Atomic transactions*: Morgan Kaufmann, 1994.

[111] S. Cherem, T. Chilimbi, and S. Gulwani, “Inferring locks for atomic sections,” *ACM SIGPLAN Notices,* vol. 43, no. 6, pp. 304-315, 2008.

[112] M. Méndez-Lojo, D. Nguyen, D. Prountzos *et al.*, “Structure-driven Optimizations for Amorphous Data-parallel Programs,” 2010.

[113] P. Blevins, and C. Ramamoorthy, “Aspects of a dynamically adaptive operating system,” *IEEE Transactions on Computers,* vol. 100, no. 25, pp. 713-725, 1976.

[114] A. Glassner, *An introduction to ray tracing*: Academic Press London, 1989.

[115] R. Ibánez, and B. Center, "Task chunking of iterative constructions in openmp 3.0."

[116] B. Jacobs, T. Bai, and C. Ding, “Distributive Program Parallelization Using a Suggestion Language,” 2009.

[117] M. Prabhu, and K. Olukotun, "Exposing speculative thread parallelism in SPEC2000." pp. 142-152.

[118] K. Olukotun, L. Hammond, and M. Willey, "Improving the performance of speculatively parallel applications on the Hydra CMP." pp. 21-30.

[119] S. Chen, P. Gibbons, M. Kozuch *et al.*, "Scheduling threads for constructive cache sharing on CMPs." p. 115.

[120] S. Hummel, E. Schonberg, and L. Flynn, "Factoring: A method for scheduling parallel loops," *Communications of the ACM,* vol. 35, no. 8, pp. 101, 1992.

[121] E. Raman, R. Rangan, and D. August, "Spice: speculative parallel iteration chunk execution." pp. 175-184.

[122] M. Cintra, J. Martínez, and J. Torrellas, "Architectural support for scalable speculative parallelization in shared-memory multiprocessors." pp. 13-24.

[123] J. Cheriyan, and S. Maheshwari, "Analysis of preflow push algorithms for maximum network flow," *SIAM J. Comput.,* vol. 18, no. 6, pp. 1057-1086, 1989.

[124] P. Diniz, and M. Rinard, "Lock coarsening: Eliminating lock overhead in automatically parallelized object-based programs." p. 285.

[125] D. Schmidt, M. Stal, H. Rohnert *et al.*, *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*: Wiley, 2000.

[126] D. Dig, R. Johnson, and M. Snir, "N-Body Pattern Language."

[127] C. Fonlupt, P. Marquet, and J. Dekeyser, "Data-parallel load balancing strategies," *Parallel Computing,* vol. 24, no. 11, pp. 1665-1684, 1998.

[128] G. Rünger, and M. Schwind, "Parallelization Strategies for Mixed Regular-Irregular Applications on Multicore-Systems." p. 388.

[129] F. Budinsky, M. Finnie, J. Vlissides *et al.*, "Automatic code generation from design patterns," *IBM Systems Journal,* vol. 35, no. 2, pp. 151-171, 1996.

[130] D. Loveman, "Program improvement by source-to-source transformation," *Journal of the ACM (JACM),* vol. 24, no. 1, pp. 145, 1977.

[131] G. Kiczales, J. Lamping, C. Lopes *et al.*, "Aspect-oriented programming," Google Patents, 2002.

[132] M. Cole, *Algorithmic skeletons: structured management of parallel computation*: Pitman, 1989.